UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Procedural modeling of cities with semantic information for crowd simulation

Master Thesis Report

Student:
Otger Rogla Pujalt

Advisor:
Nuria Pelechano Gómez
Department of Computer Science
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Co-Advisor:
Gustavo Ariel Patow
Department of Computer Science, Applied Mathematics and Statistics
Universitat de Girona (UdG)

Master in Innovation and Research in Informatics
Specialization in Computer Graphics and Virtual Reality

Date of defense: 4 July 2016

Resum

En aquesta tesi de màster es presenta un sistema per a la generació procedural de ciutats poblades. Avui en dia poblar entorns virtuals grans tendeix a ser una tasca que requereix molt d'esforç i temps, i típicament la feina d'artistes o programadors experts. Amb aquest sistema es vol proporcionar una eina que permeti als usuaris generar entorns poblats d'una manera més fàcil i ràpida, mitjançant l'ús de tècniques procedurals. Les contribucions principals inclouen: la generació d'una ciutat virtual augmentada semànticament utilitzant modelat procedural basat en gramàtiques de regles, la generació dels seus habitants virtuals utilitzant dades estadístiques reals, i la generació d'agendes per a cada individu utilitzant també un mètode procedural basat en regles, el qual combina la informació semàntica de la ciutat amb les característiques i necessitats dels agents autònoms. Aquestes agendes individuals són usades per a conduir la simulació dels habitants, i poden incloure regles com a tasques d'alt nivell, l'avaluació de les quals es realitza al moment de començar-les. Això permet simular accions que depenguin del context, i interaccions amb altres agents.

Resumen

En esta tesis de máster se presenta un sistema para la generación procedural de ciudades pobladas. Hoy en día poblar entornos virtuales grandes tiende a ser una tarea que requiere de mucho tiempo y esfuerzo, y típicamente el trabajo de artistas o programadores expertos. Con este sistema se pretende proporcionar una herramienta que permita a los usuarios generar entornos poblados de un modo más fácil y rápido, mediante el uso de técnicas procedurales. Las contribuciones principales incluyen: la generación de una ciudad virtual aumentada semánticamente utilizando modelado procedural basado en gramáticas de reglas, la generación de sus habitantes virtuales utilizando datos estadísticos reales, y la generación de agendas para cada individuo utilizando también un método procedural basado en reglas, el cual combina la información semántica de la ciudad con las características y necesidades de los agentes autónomos. Estas agendas individuales son usadas para conducir la simulación de los habitantes, y pueden incluir reglas como tareas de alto nivel, la evaluación de las cuales se realiza cuando empiezan. Esto permite simular acciones que dependan del contexto, e interacciones con otros agentes.

Abstract

In this master thesis a framework for procedural generation of populated cities is presented. Nowadays, the population of large virtual environments tends to be a time-consuming task, usually requiring the work of expert artists or programmers. With this system we aim at providing a tool that can allow users to generate populated environments in an easier and faster way, by relying on the usage of procedural techniques. Our main contributions include: a generation of semantically-augmented virtual cities using procedural modeling based on rule grammars, a generation of a virtual population using real-world data, and a generation of agendas for each individual inhabitant by using a procedural rule-based approach, which combines the city semantics with the autonomous agents characteristics and needs. The individual agendas are then used to drive a crowd simulation in the environment, and may include high-level rule tasks whose evaluation is delayed until they get triggered. This feature allows us to simulate context-dependant actions and interactions with other agents.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the recent advances in low-cost virtual hardware and high performance graphics, virtual reality (VR) has moved from the research lab into the general public. Large virtual environments can be found easily on the Internet, and can be visualized with inexpensive VR setups such as mobile phones with a physical device (a cardboard or plastic glasses). The one thing missing in most virtual environments are believable human figures wandering around in a realistic manner. And the main reason for this is that there is still a lack of tools to populate such environments in an easy and semi-automatic manner.

Typically we can create models for virtual humans with software tools and services such as Mixamo, MakeHuman or Autodesk Character Generator to mention a few. This software allows us to obtain a rigged and textured skeletal mesh along with a variety of animation clips. However, the process of getting those virtual humans fully integrated in an environment, being simulated and showing a purpose as they wander the virtual worlds, is still a challenge.

In recent years we have observed that both video games and movies are showing increasingly larger crowds populating their scenes. In most cases, those scenarios are the result of either (1) an automatic process to generate scenes of short duration where the agents only follow some predefined trajectory or simple behavior, or (2) the work of a dedicated team of expert artists, story writers and animators working for months to achieve the desired virtual population.

It is thus necessary to explore new research areas that will allow us to develop tools that can ease this process. In this master thesis we investigate procedural models as a way to rapidly generate large cities augmented with semantic information, which can be then used to drive the behavior of a virtual crowd generated to populate such environment.

For this thesis we have developed a full prototype of such system, which implements all the different parts required for the generation of a city and its population. This prototype contains the core features, and has been engineered to be easily extended, aiming at the addition of further features and options as future work.

1.2 Objectives

The main goal in this project has consisted of developing a tool to ease the process of authoring populated virtual cities, by using procedural generation methods. This general goal has been divided in the following subgoals:

- To generate a basic procedural city augmented with semantic information such as land usage or building types, and interactable objects.
- To investigate a public database of real world population data and extract meaningful statistics that can be used to generate a more realistic and diverse city and crowd.
- To automatically generate a population consisting of virtual agents based on the information extracted from such data.
- To generate *agendas* for each virtual human with procedural methods. The agendas represent a high-level list of their daily activities such as going from home to their workplace and back. In addition, they may include low-level context-dependant tasks whose evaluation is deferred to execution time.
- To drive the virtual humans' simulation following their generated agendas, expanding the deferred tasks as required.
- To develop and integrate this system in an existing 3D engine, in order to benefit from its features and avoid the need to deal with matters such as asset loading, animation handling, and model rendering, among others.

1.3 Contributions

The main contribution of this work is a prototype of a system integrated into Unreal Engine 4, and aimed at procedurally generating populated cities with agenda-driven inhabitants. It is formed by three main modules that handle different parts of the generation and the simulation. We will now describe briefly each of those modules:

1. The first module generates the layout of the city, which corresponds to its building lots, either by loading real-world data or by running a basic isle generation algorithm. This module provides the basis for the other two, but due to the large scope of the project, the main focus has been shifted into the other two modules as they were deemed more significant. Therefore, there is room left for implementing more rich generations as future work.

2. The second module is aimed at the generation of the actual building geometry in the city lots, and implements a modified subset of *Computer Generated Grammar* (CGA), which consists on a rule system that uses user-specified grammars to generate geometry. In addition, this rule system has been extended in order to allow the generation of some additional semantic data such as interactable entities, or entry points for buildings.

3. A third module deals with the generation of the population, and then the simulation of their whereabouts during a day. The generation is performed using real-world statistical data, and for each person a high-level agenda is generated using a rule system of similar syntax but different to CGA. This system also allows specifying rules of deferred evaluation for lower-level tasks. These agendas are then executed by the virtual agents, in a full-day cycle.

1.4 Organization

The rest of this work is structured as follows. In the next chapter, the state of the art on some different topics related to the work in hand is discussed. In chapter 3, the developed work and the implemented features are detailed, along with some examples, while also explaining each step of the generation. Chapter 4 shows some of the results obtained with this system, and finally in Chapter 5 conclusions are exposed along with the current limitations of the system and future work.

Chapter 2

State of the art

The work on hand involves several different fields and subfields. Therefore, a discussion of the state of the art in the most closely related topics is presented.

2.1 Crowd simulation

The crowd simulation field focuses on how to simulate relatively large groups of virtual agents and their behaviors in a realistic way. There has been a large amount of work on this field in the last decades, aimed to achieve plausible simulations for movies, video games, evacuation planning, training, etc.

Crowd simulation models are usually divided into two types: macroscopic and microscopic. The first ones focus on the crowd behavior as a whole, for example considering global movement flows; while the latter ones focus on achieving a more individualized behavior of the agents, usually by considering only other agents or objects in its near vicinity.

There are several areas that deal with different aspects of the simulation of crowds: the navigation of the agents, the animation of each individual, the rendering of large numbers of agents, the generation and authoring of crowds, etc. In the next subsections the state in some of the most relevant areas is discussed.

2.1.1 Navigation

One of the main issues in the simulation of crowds, if not the central one, corresponds to achieving a successful and human-like movement behavior for the agents.

There are numerous works proposing approaches for the navigation-related local interactions of the agents, notably the collision avoidance with other individuals or obstacles, but also other behaviors such following or running from other agents. These kind of models can be categorized as microscropic models, as they take into account each agent individually.

This navigational emergent behavior goes back to Reynold's *boids* [Rey87], which proposed a system of 3 rules (figure 2.1) to keep the cohesion of a group of agents while avoiding collisions among them, achieving a behavior similar to that of a flock of birds or school of fishes.

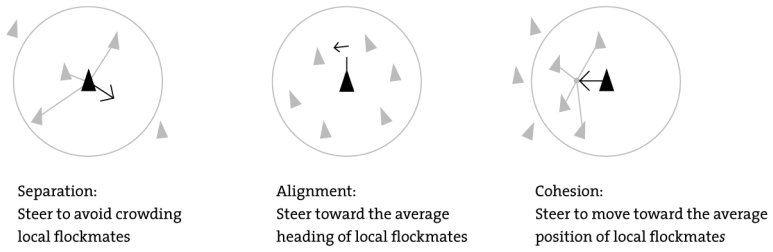

Figure 2.1: The three rules in Reynold's Boids

Since then, many other works proposed ways to simulate behaviors for navigation. Some of the most popular or approaches are:

- Rule-based systems, which use rules to specify the behavior of each agent or groups of them. One example of this is the work by Musse and Thalmann [MT01], which for instance allows specifying rules for agents currently classified with a hungry status to direct them to pick up food from a table.
- The social forces model [HFV00], which uses a variant of the Newton's second law of motion from physics ($\sum F_i = m\,\Delta v$) to model the agents. For each agent, a sum of the forces affecting it is computed, which is then used to determine the agent velocity vector. Some possible forces are repulsion (avoid obstacles), or attraction (head to a goal), but many others can be included (e.q. for queueing behaviors).

- Velocity approaches, such as the popular Reciprocal Velocity Objects (RVO) model [VLM08], which computes the optimal velocity that minimizes a certain penalty metric. This penalty considers the deviation from the preferred velocity as well as the expected time to collision. A Monte Carlo sampling is then performed on the space of candidate velocities to pick specific values.
- Hybrid models, like HiDAC [PAB07], or Hybrid Long-Range Collision Avoidance [Gol+14], which combine some of the previous methods, in order to overcome the individual limitations of each one.
- Least effort methods, such as PLEdestrians [Guy+10], that resort to optimization techniques to compute an optimal velocity according to some metrics based on biomechanical principles.

There are also proposals of macroscopic models that try to handle all the agents simultaneously, which tends to perform better for high-density situations. For example, cellular automata models such as in the work by Chenney [Che04], which uses tiling techniques to build a regular grid representing a vector field of velocities. Another proposal is the one by Narain et al., that combines discrete agents with the specification of a continuous system that considers the incompressibility of the crowds to perform better in high densities [Nar+09].

More recently, data-driven models have been proposed. For instance, video footage of real crowds can be used to extract data, that is in time used to determine the navigation patterns of the crowd [Cha+14]. One of the advantages of these approaches is that the data can be used to validate the results.

2.1.1.1 Goal-oriented navigation

A particular case of the crowd navigation problem is when each agent or group of agents are assigned a specific target location to reach through their movement. Typically in this case a global navigation algorithm is employed, which provides a preferred movement direction and/or velocity for the agent; and then a local navigation algorithm such as one of the previously described ones is used to compute a corrected velocity taking into consideration behaviors such as collision avoidance, keeping formations, etc.

For the global navigation part, a common approach is using a path-finding algorithm such A* or one of its variants (see [RC10] for a survey) over a navigation

map. The main problem of this is that the execution may become slow for large crowds or complex environments, or that it may be necessary to adapt the paths for zones with a high-density of agents where the local navigation algorithms are unable to resolve collisions well on its own.

Another proposal for global navigation are Navigation Fields [Pat+11], which are defined as a 2D uniform grid representing a field of preferred velocities for the agents, such that each agent velocity is defined by its nearest cells. This provides a per-group goal-oriented global navigation, but still requires of an underlying local navigation algorithm to avoid collisions. Also, it assumes there are relatively large groups of agents with a common goal, so it is most useful in cases such as to form movement lanes in narrow areas.

2.1.2 Agent personality and interactions

A large portion of the works on crowd simulation focus in the navigation problem, but in order to achieve more realistic simulations the inclusion of further individualized behaviors is usually required.

Some works propose gestural interaction models between humanoids, or humanoids and simple objects in their surroundings which have been previously manually tagged and classified [Kap+15]. This topic of interaction with semantically-enriched objects is discussed in more depth in a subsequent section.

Badler et al. proposed a parametrization of agent actions [SZP00], which consists on rules parametrized by the participants (objects and agents), and was employed in order to translate high-level orders in natural language (e.g. "sarah, walk to the door") into low-level tasks. Later on, this representation was integrated with a personality and emotions model for the agents [AB02], achieving more individualized animations and actions. To represent the personality, the popular Five Factor or OCEAN model [Dur+08] was used (openness, conscientiousness, extroversion, agreeableness, and neuroticism).

Allbeck continued in this line of work proposing Functional Crowds [All10], in which the agents *inhabit* the space as opposed to passing through it. It uses parametrized actions associated to both roles and groups, and employed the tasks,

contacts and calendar features in the application Microsoft Outlook© to build high-level agendas of activities to perform during the day, which averts specifying each low-level action or each agent manually, while also accounting for some level of randomness and hence variation.

In another direction, reinforcement learning (RL) approaches have been also proposed in order to achieve more adaptable individual behaviors [MLF15], but these techniques are often difficult to control and adapt due to them resorting to use reward functions.

Finally, some works try to combine the individual behavior with per-groups behaviors, providing hybrid macroscopic-microscopic models. For example, using a system that allows defining behaviors for groups that can be refined by more individualized ones [MT01].

2.1.3 Authoring of crowds

Even with a realistic simulation model, one of the main difficulties for obtaining a simulation that suits the user needs tends to be the amount of work required in tuning the simulation parameters and/or the characteristics of each agent. This difficulty is becoming more of an issue as better but more complex simulation models are proposed, so works on this topic are still relatively scarce. Most of these works aim to provide quick and easy forms for the user to specify trajectories, densities, etc.; some even allowing editing the simulation during its execution.

For the previously mentioned work on Navigation Fields [Pat+11], a system to allow the user to sketch user directions is implemented, allowing some agent groups to reach their specified goals but deviating to follow the user-specified paths at the same time. A similar but more general proposal also allows to define relative positions or velocities (e.g. to achieve formations) for the crowds to follow [ACC14], while also allowing defining movement fluxes in form of velocity fields.

Crowd Patches [Jor+14] [Jor+15] are another authoring alternative. In this case, the user can connect and deform a series of polygonal patches with predefined movement paths and entry/exit points to fill the desired space, obtaining a quick an easy way to populate an environment. An example of the results can be seen

in figure 2.2. Another variant of this technique allows the user to define a crowd density field [JPC14], from which Crowd Patches are computed.

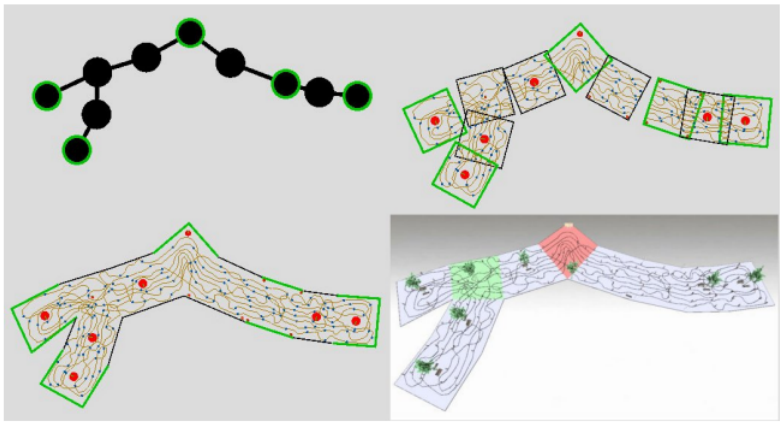

Figure 2.2: Example of crowd patches

Another proposal somewhat similar to the previous one uses cage-based deformation [Kim+14] to allow space/time manipulation of the crowd by the user, while keeping as-rigid-as-possible deformations. In general, these kind of methods allow easy and intuitive editing of crowd animations with real-time performance, but the agents are just dummies that wander in the environment with no real goal locations, or with little to none interesting behaviors and relationships among them or with the environment.

2.1.3.1 Authoring in commercial applications

There are some commercial applications that focus on providing crowd simulation and authoring, but are usually very focused to the needs of the film industry, and hence not designed for real-time execution in mind.

Some commercial tools such as Maya [Aut16b] and 3ds Max [Aut16a] have incorporated procedural models for the generation of animations, but not account for the global simulation of behaviors in the agents or their evolution in time. Furthermore, these models provide short animations that are only acceptable for small time intervals (seconds or minutes at most), as otherwise repetitive behaviors or animations become evident. For long sequences, they offer simulations for

navigation but using basic locomotion animations for the characters.

For Maya, there are also commercial plugins focused on this topic, which provide an easy way to generate crowds and their behaviors, in some cases through procedural generation [Bas16] [Gol16]. They have been used extensively to simulate crowds such as armies or zombie swarms for films.

2.2 Virtual cities

The other main related fields correspond to the generation of virtual cities in both its geometry and enrichment with semantic data, and how to populate this kind of environment with virtual agents; topics which are discussed in the following subsections.

2.2.1 Geometrical modeling and procedural generation

Procedural generation of content is a field that has been growing in interest the recent years, motivated in great part by the film and videogame industry. Every year the processing and rendering capabilities of computers increase, which allows for larger quantities and variety of content, or higher detail; both of which require investing more time and effort into developing such content. Procedural methods, although they tend to require a bigger initial effort, tend to reduce the time investment at long term. Hendrikx et al. [Hen+13] discuss in a survey several procedural generation methods for different kinds of content or aspects, and provide some examples in the field of video games. The discussed content types include generation of urban environments and buildings.

Procedural modeling has been used with success for both urban and non-urban environments, generating very rich geometry requiring only a very small effort and time from the user. One of the most prominent approaches for the procedural generation of cities consists on the creation of a 2D of map city (with building lots, network of roads, etc.), which is subsequently followed by a procedural generation of the buildings on the lots described in it.

On the generation of the exterior geometry of building, a very powerful procedural

approach and very likely the most popular one is Computer Generated Architecture (CGA) [Mül+06], that defines a rule system based on parametrized grammars (*shape grammars*). It allows writing rules on how to generate the buildings, refining the details through grammar productions, and allowing for parameters and the use of pseudo-random values to generate variations. This method has been subsequently improved (CGA++, etc. see [SW15]), and it being used in the commercial software CityEngine [Esr16]. Figure 2.3 shows some of the results that can be achieved through this method.

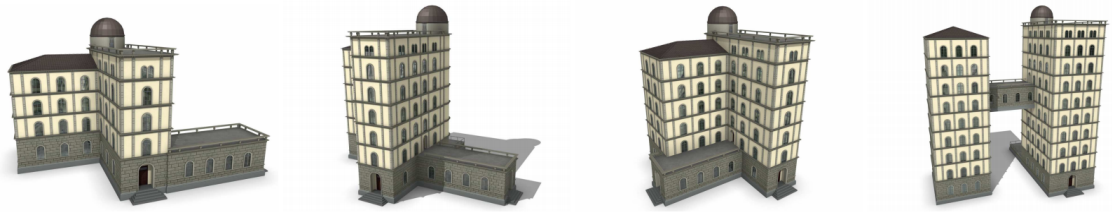

Figure 2.3: Examples of buildings generated procedurally with CGA

The generation of the roads is usually performed by means of a road network (a graph) [PM01], which implicitly also defines cells that can subdivided pseudo-randomly and used as building lots. Kelly and McCabe provide a survey [KM06] on the generation of city geometry, including different ways to generate road networks (L-systems, agent simulation, template-based generation, etc.). Figure 2.4 shows a possible result of one of such processes.

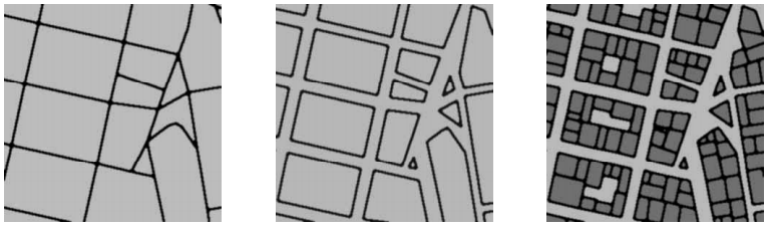

Figure 2.4: Example of the generation of city lots from a road network

There is also work on the topic of procedural generation of non-urban environments, which may provide ideas that may be adapted. One of such works uses statistical information to allow the user to create outdoor terrains by learning the distribution of trees, grass, rocks, etc. constrained by the terrain slope [Emi+15]. The user can then paint and populate new terrains copying the object distribution.

In recent work, Feng et al. [Fen+16] present a system that starting from a boundary polygon (e.g. for a shopping mall), generates paths and sites optimizing three crowd flow properties: mobility, accessibility, and coziness. Their approach is based on training nonlinear regressors using real data, which learns the relation between those agent-based metrics and the geometrical and topological features of the layout. Some of their results are shown on figure 2.5.

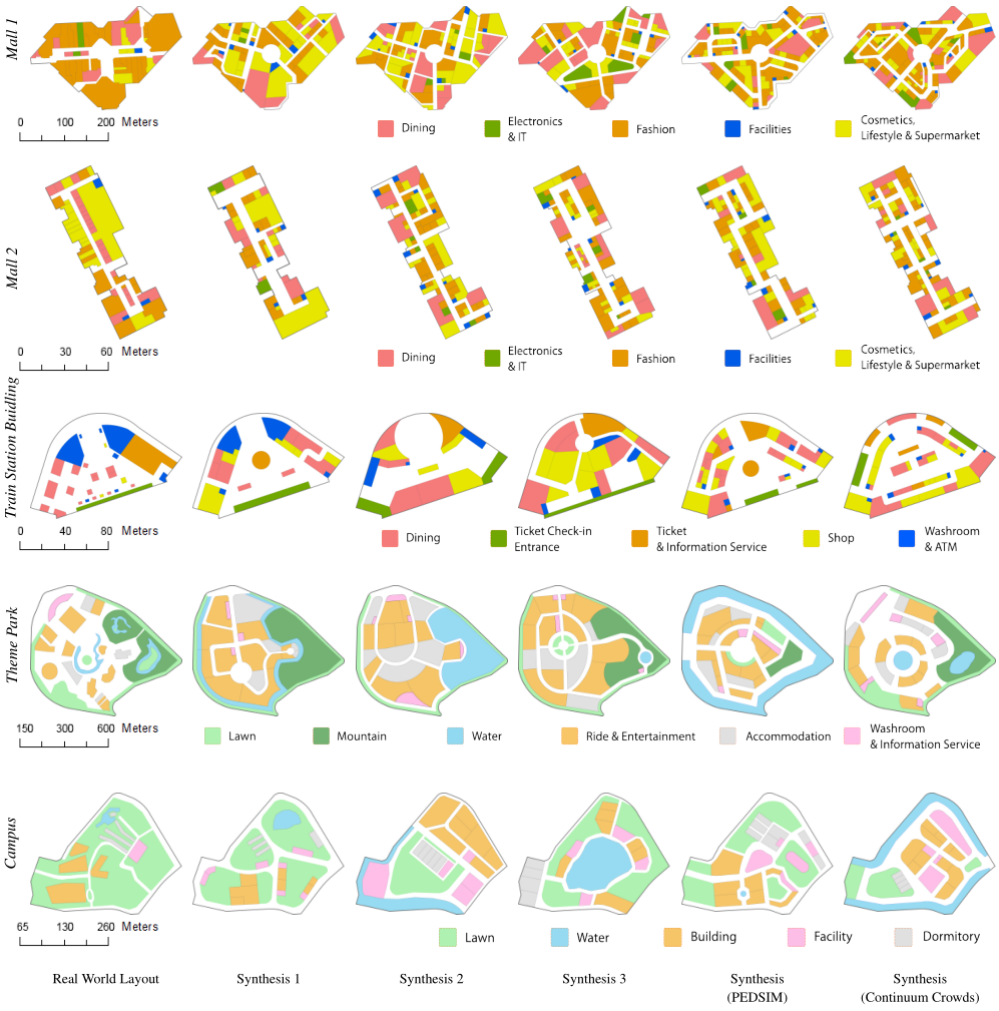

Figure 2.5: Layouts generated by optimizing crowd flow properties

2.2.2 Behavioral urban modeling

One step further from the geometric urban modeling which is purely computer-graphics oriented, is the behavioral urban modeling. The aim of this area is to obtain meaningful predictions of the socio-economical evolution of an urban area [AWS08] [Wad02], i.e. how a city will change over time. These simulation models, however, tend to use limited and fixed 2D geometric features (e.g. regular grids, or parcels), and are computationally too expensive for real-time simulations.

Some work has been done with the aim of obtaining more general 2D or 3D city layouts that change over time [Hon+04] [Van+09]. However, those methods merely compute the changes on a model, but do not attempt to explain them.

Weber et al. proposed a method to create 4D cities [Web+09], i.e. interactive 3D urban environments that change over time. The geometry of those cities is created using rule systems, and is determined by decisions for street expansions, traffic densities, land use simulation, lot subdivision, etc. A simulation of the traffic is used to establish the amount of people using a street segment per day, information which is then used to compute street widths and land usages. The proposed model is stochastic and uses sampling over a discrete number of trips in the street network, but is limited to medium-sized cities. Figure 2.6 shows an example of the evolution.

Another approach by Vanegas et al. [Van+09] subdivides the urban space into a regular grid of cells, each one of which is assigned a set of variables that control the distributions of population and jobs, land values, road network, parcel shape and building geometry. These variables can be globally or locally modified or constrained. Their behavioral framework –based on a simplification of Urban Sim [Wad02]– and their geometrical modeling engine produce a single dynamical system that seeks a behavioral and geometric equilibrium after each user-specified change, but which lacks of elements of behavioral modeling such as the capacity of prediction.

2.2.3 Enrichment with semantic information

To be able to (semi-)automatically populate a city, apart from the generation of the city geometry, additional information is required: for instance, the lots may

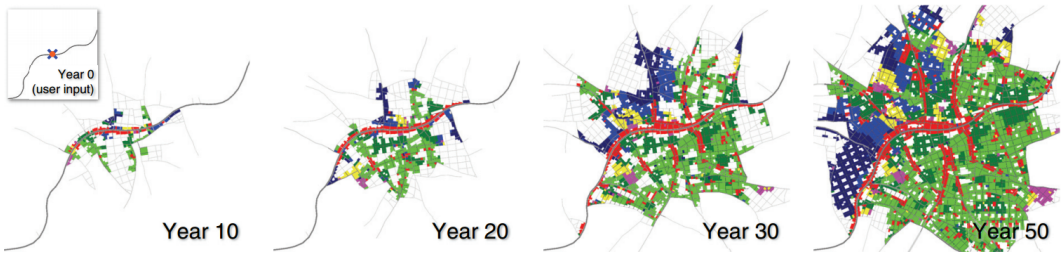

Figure 2.6: Simulation of the evolution of a city, where the colors correspond to land uses.

be tagged with semantic information such as the possible usages of the buildings by the virtual citizens, the services they offer, or their capacity of people.

There are some works that deal with assigning *affordances* to virtual objects. One example of this by Pelkey and Allbeck [PA14] focuses on semi-automatically assigning semantic affordances to objects. However, most semantic tagging of objects still requires some degree intervention of the user and tagging by hand.

A work by Hocevar et al. deals with the case to model procedural behaviors for groups of agents, given an environment where entities have been enriched with semantics [Hoc+12]. This work also uses as a concept the *personal space* to avoid collisions, and allows for the emergence of behaviors such as formations.

For cities, a typical strategy is to include as part of the procedural generation of the lots semantic information (e.g. land use) that will then condition the kind of building it will be built on it. This information can be distributed according to user parameters or some statistical data, or can also obtained for instance through a more advanced simulation such as the 4D city example in figure 2.6 [Web+09].

2.2.4 Population with virtual inhabitants

A first step required to integrate the simulation of agents into a city consists on the generation of a navigation map from the city geometry, which will then allow to execute path finding algorithms that direct the agents. On this line of work, there are several proposals, from a basic polygonal navigation mesh to a more advanced structure [Kap+13].

Some works have studied how to use geolocalized data from social networks to populate virtual environments corresponding to real locations. For example Bulbul et al. extracted data from Twitter and Instagram, and used maps from OpenStreetMap, to simulate pedestrians [BD16], which is shown in figure 2.7. The limitation of this approach is that the positions may be not representative of the real crowd density of the zones, but more biased towards the points of interest; and also that considering the movement over time, the locations might be very undersampled or they might only be a few usable instances in the data.

Figure 2.7: Simulation of a street of real city using data from social networks.

There are some works that populate cities using simple state machines that launch basic animations at certain times or places, to achieve more realism, such as the commercial software Massive [Mas16], but they do not allow editing the behavior so that it is defined by the interaction with the environment, just as the real humans use their surroundings. The method of Crowd Patches mentioned in a previous section has also been used successfully to populate virtual cities [JPC13], but also suffers of this problem.

In the topic of populations that do interact with their surroundings, one of the most notable works is by Jorgensen [JL14] [Jor15], which proposes a system of task scheduling for agents in an urban environment where areas are semantically tagged. One of the main limitation of this method is the size of the environment, as it provides a very detailed simulation that includes building interiors (see figure 2.8). Also, while it provides means for the simulation, the used environments are static models where the zones have been tagged by hand. In addition, agents do not change their behaviors over time or learn from each other, and the tasks they are to perform need to be specified beforehand. On the other hand, it accounts for agents personalities and preferences, and also includes cooperative tasks.

On a related topic, as the simulation of large populations may be very costly, there

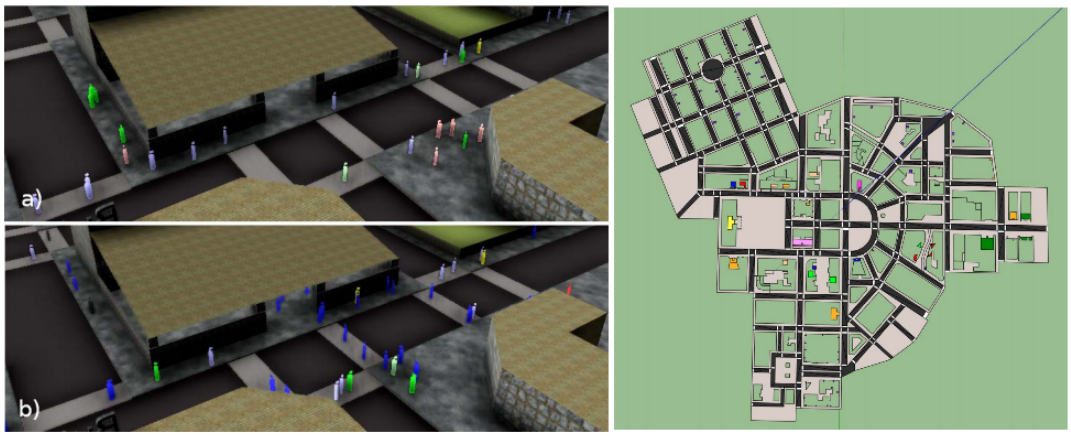

Figure 2.8: A populated urban environment. Left: a school at (a) 4:10 and (b) 4:20 PM, notice the flow of blue people. Right: the informed environment used in the simulation.

are some proposals on how to reduce the amount of simulation required. One example of this is the work by Sunshine and Badler, which propose to only run the visible subset of agents, while using generated *alibis* based on statistical techniques in order to guarantee a correct long-term simulation (e.g. agents entering/leaving the zone) [SB10].

2.2.5 Real-time visualization

A final topic to be briefly discussed is the visualization of the populated cities. Although the rendering of a city may be optimized by using specific algorithms that take advantage of its nature as a 2.5D scene (2D + height), this lies outside of the scope of this project. More general-purpose rendering optimizations such as geometry-culling techniques, or level-of-detail (LOD) models for far buildings and agents may be considered, as they are already implemented on most rendering engines.

For procedurally grammar-generated buildings, there has been work on how to automatically generate textured LODs for the resulting geometry [BP13], including the generation of several models depending on the point of view, as is shown in figure 2.9.

As for the rendering of large amounts of agents, there are also many works, most

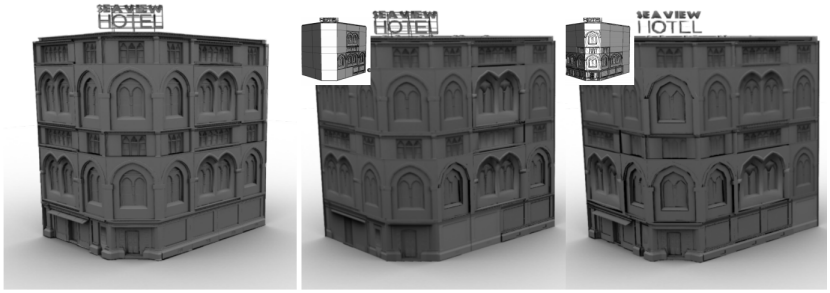

Figure 2.9: Two LOD models created for a procedurally generated building (original in the left)

of which go in the direction of using impostors for the farthest individuals and LODs for the medium range ones, the main difficulty being the handling of the animations, which are usually also simplified or even precomputed into impostor textures [Bea+11] [BP14]. In addition, to obtain a greater variety of different agents usually tricks such as using different clothing/skin textures are employed.

Chapter 3

Architecture

3.1 Overview

The developed framework requires many different subsystems, therefore the implementation has been divided into different modules in order to keep the dependencies low and offer reusability of code. Figure 3.1 shows an overview of the different parts of the system, their dependencies, and how they interact with each other.

The city generation is divided in two modules: generation of the layout of lots, and then generation of buildings in the lots. For the first, two options have been implemented: a very simple isle-like generator, and a real-world map data loader. For the generation of buildings, a custom implementation of CGA (Computer Generated Architecture) is used, which generates building geometry by executing a grammar of rules, but also augments the results with semantic data.

A third module contains the city population generation and behavioral simulation. It creates a set of households ("families") with members of various ages and genders, and then generates their agendas using a system of rule grammars similar to but different from CGA.

These three modules are designed separately and using the engine world as the main method of communication, as this provides greater flexibility, by allowing the user to perform modifications by hand after the generation, or experimenting in each step with different parameters until the desired results are achieved.

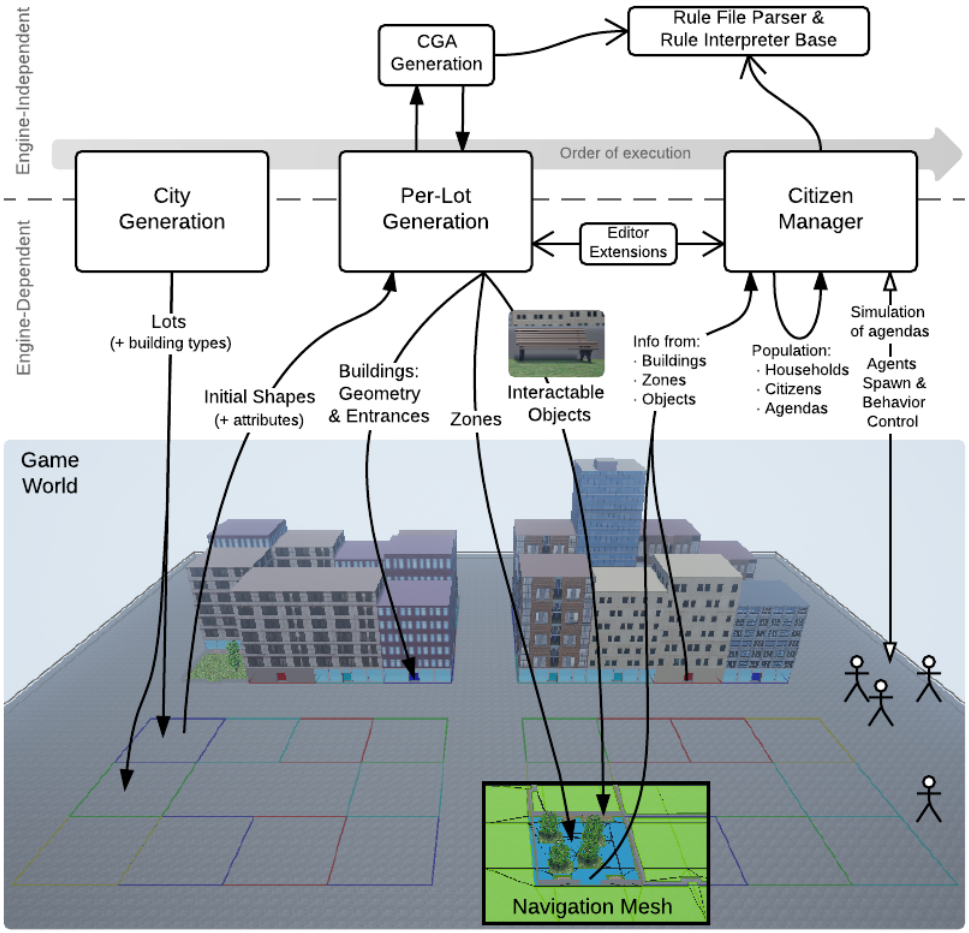

Figure 3.1: Overview of the developed system and its modules. Filled black arrows represent the module inputs and its results, and open arrows indicate dependencies.

The generation of lots in the first module creates as a result game objects in the world, which contain the polygons that represent the footprints of the lots. They are also assigned a lot or building type that will determine what will be built there.

These lot game objects are actually instances of a CGA Initial Shape class, which ties in the CGA generation module with the engine. It provides options for generating or clearing the geometry, and modifying the CGA attributes values (external parameters used in the execution). And in fact, the previously mentioned building

types are assigned as one of those attributes. Running the generation in one of those game objects results in the creation of game object components, which are attached to it. These components can be of several types: a procedural mesh component for the generated geometry, a navigation mesh modifier for specified zones, or a custom component for building entry points specification, among others.

As for the population generation and simulation, they both are executed when the game is started, as opposed to the previous two modules which run inside the editor mode. A singleton game object, instance of a class that has been named *CitizenManager*, performs this generation at its initialization, while its *update* function controls the simulation by keeping track of the current time of the day and updating the agents' agendas and behaviors.

At the start of the initialization of this class, a small preprocess step gathers the information about the city and the environment into a helper class, which will be used to find objects or answer queries later on. To carry out this step, all the game objects in the world are iterated in search for specific components (e.g. the building entry points), thus reading the results generated by the previous module. In addition, the navigation mesh is also processed, performing a search to find the polygons of zones of interest that touch zones of another type, hence locating the zone entrances.

Apart from the CGA generation module, the rule file classes and interpreter base, which are fully independent from the engine code, all other modules depend on it at different degrees. Nonetheless, they have been designed such that they could be easily adapted to be used by other engines. The only exceptions would be the extensions to the editor, which as expected is mostly engine-specific code. These editor extensions contain debugging or visualization features, so while they can aid in the development and the debugging of the generations, they are not essential for the system operation.

3.2 Engine and tools used

3.2.1 Choice of an engine

In order to benefit from the state-of-the-art technologies on 3D rendering and avoid reinventing the wheel and having to directly handle the asset loading, the animation systems, etc., the logical choice is to use an existing engine or tool which has been extensively optimized for such purposes. Among these, there are several commercial video game engines that are offered free of cost under certain conditions.

One of the most popular engines is Unity 3D [Uni16], which supports programming in C# and Javascript, as well as modules in C++ in form of dynamic libraries (DLL).

An alternative to Unity 3D which is gaining popularity is Unreal Engine 4 [Epi16]. It presents a similar feature set, but supports programming in C++ as main language, and also allows us to access and modify the source code of the engine, hence allowing the programmers to tune the engine depending on their needs.

Other alternatives include CryEngine [Cry16], or the Source engine from Valve Software [Val16]. However, they have been not considered for various reasons. In the case of Source, because although development is also in C++, the engine is somewhat outdated and is expected to be replaced soon by its successor Source 2. In the case of CryEngine, while having similar capabilities with Unity and Unreal, it has a very steep learning curve, and is more suitable for large development teams. And in addition, its community is smaller and there is fewer documentation. For this same last reason, other less known game engines have not been considered either.

Regarding the economic cost of the engines, all the previous allow free usage for the development, as despite they are not free, they resort to either a royalty-based system in case of commercialization of the resulting product, or a limit for the revenue under which no money is demanded. In the case of Unreal Engine, they ask for a 5% royalty on the collected money after the first 3000 USD per quarter. For Unity, the free version is free of cost for companies with a yearly gross revenue inferior to 100.000 USD, or the same quantity as budget for non-profit entities.

Our choice was therefore narrowed down to Unity or Unreal. Both of these engines come with a custom editor application used to edit the virtual worlds, place objects in them, and control and edit their properties. The engines internals are also quite similar, as they use component-based object architectures, where each world object contains a list of attached components, which is the tendency on game and VR development as it allows a great deal of flexibility and reuse of code.

Unreal Engine 4 has been chosen in the end for the following reasons:

- As indicated earlier, it provides access to the engine code and allows the programmer to modify it when needed, whereas in Unity the engine is a black box. Hence, in the long term if specific tunings are needed for example to optimize crowd movements by considering global solutions, they can be applied easily.
- As it uses C++ natively, third party libraries can be easily integrated with the project. Furthermore, the code resulting from the project can also be exported with very little work for its usage in other systems, whereas that would be more difficult if written in Unity-specific C# or JavaScript. Unity does also support C++ code in form of dynamic libraries (DLL), but this is designed for external libraries and the access to the engine is more convoluted, requiring in addition the writing of wrapper classes in C++/JavaScript. Moreover, it does not support DLL hot-swapping unlike Unreal Engine.
- It offers some more advanced state-of-the-art features and potential at long term, and the non-written consensus is that it is "one step further" than Unity in rendering capabilities, which is supported and explained by the fact that it has been and is used to develop high-budget videogames.

The main drawback on this choice was the fact that Unreal Engine 4 is regarded as having a more pronounced learning curve, resulting in some slowing down in the project development. However, as the project was aimed to be continued afterwards in further research, it was deemed as the best option in the long term and hence worth a slow start.

3.2.2 Generation of animated human models

In order to generate rigged 3d skeletal models for the virtual humans, two applications free of cost have been considered. Both present similar features, allowing

selecting among different variants for each body part and customization through tuning sliders, but there are some particularities. The two applications evaluated and tested are:

- *Mixamo Fuse* [Mix16a]: a desktop application by Mixamo (Adobe) that allows directly choosing a mesh for each body part, as well as hair, clothing, etc. The features of each part can be then very fine-tuned through sliders (e.g. separation of the eyes, shape and size of nose, mouse, ears, etc.) and even the clothing can be modified by choosing among a variety of texture colors (see figure 3.2). The main issue of this program is that it generates very high quality models (i.e. models with multiple high-poly meshes and different textures), which are not very suited for the simulation of crowds since it has a big impact on rendering performance. The tool does not have an option to choose from lower resolutions or LOD generation. Mixamo offers another service called Decimator [Mix16b] which aims to generate LOD models and reduce the vertex count. The tools allows the user to choose the percentage of vertices to be reduced from the original mesh. The resulting static character offers an acceptable quality, however the problems arise when animating the character and there are many artifacts introduced in the rigging. We tested several characters with different percentages of mesh simplification and then inserted them into the crowd simulation with a variety of animations. The final results were characters with severe artifacts such as intersections of different types of textures around the joints, which were not acceptable.

- *Autodesk Character Generator* [Aut16c]: a web service which allows generation of characters by modifying each body part. A particularity is that instead of allowing a choice of each part and then slider-based tuning, it works by allowing to choose for each part feature two of the presets and then interpolating/blending them with a single slider (see figure 3.3). It is somewhat less customizable on the clothing part (only 1 choice and no tuning) and has some paid-only items, but it has the advantage that the generated models are formed by a single mesh and texture, and in the free version both low and medium quality versions can be selected.

The latter has been used in the end, as it is more suited for rendering large crowds. For the sake of an example, a normal model generated with Mixamo contained about 15k vertices, whereas with ACC (in the lowest quality) resulted in about 2.3k vertices.

However, an issue with both of the said character generation options is the lack of an easy way to generate models of children. In Mixamo there is some "toon teen"

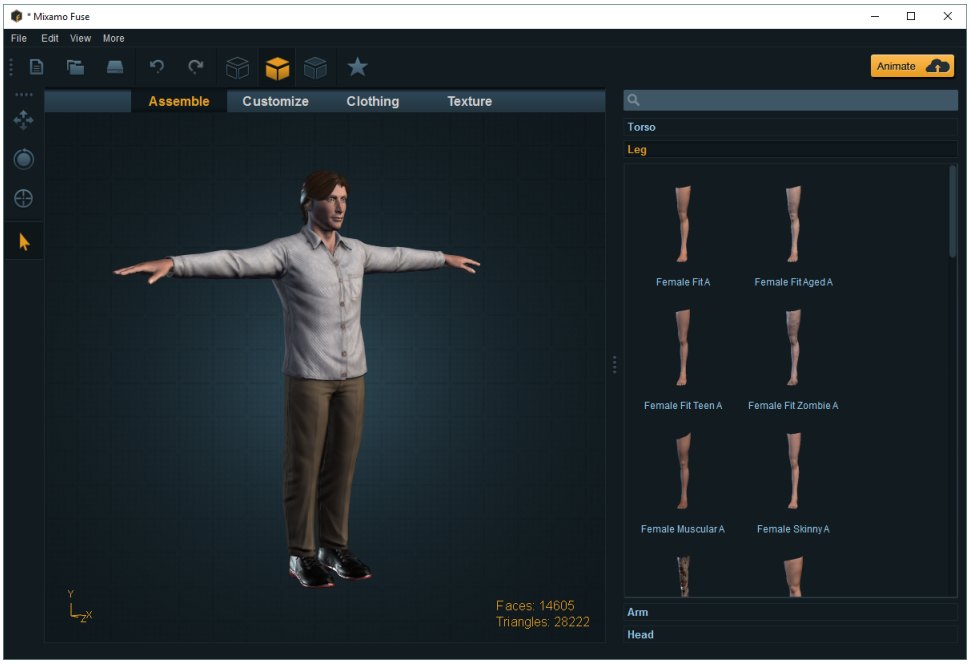

Figure 3.2: User interface of Mixamo Fuse

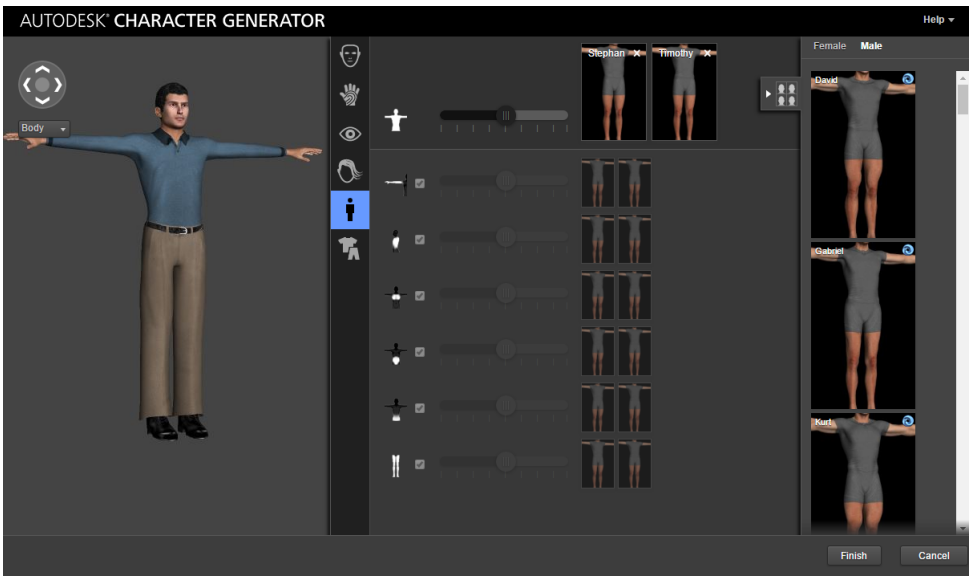

Figure 3.3: User interface of Autodesk Character Generator

models but whose style significantly differs from the rest of models as the name suggests, whereas in Autodesk Character Generator there is no direct support but only a single option to modify the height of the character. The used children

models have been generated exploiting this latter option, along with hand-tuning some of the customization sliders to achieve a kid-looking model (big eyes, less pronounced features, etc.). The result of this is far from realistic (the limbs are somewhat still disproportionate), but are sufficient for showcasing purposes.

To generate animations for the characters, the Mixamo [Mix16c] non-paid online service has been used. This web site allows uploading a rigged or non-rigged mesh, creating a rigging if necessary, and lets the user browse with an UI a relatively large variety of different animations, which can be previewed on the uploaded model where the service internally generates or retargets the animation for it. The animations can be also tuned with some parameters (e.g. distance body-arms to avoid self-collision and adapt better the animation to the model), and then can be combined in a pack and downloaded together with the model, typically in a single or multiple FBX files.

Due to time constraints, for the prototype implemented in this project only 4 different models were included: for each gender (male/female), a children and an adult version. More models could be easily added to the system, but this requires manually editing them one at a time, and then uploading them also manually to apply the animations. Also, no easy way of recoloring the textures has been found, as the textures pack both the clothing and skin.

3.3 Generation of the city

The first part of the project consisted on generating the geometry of a city to be populated, following a similar approach to the one explained in the state of the art. The overview of the process is the following one:

1. Generation of city lots, i.e. polygonal footprints of the places where a building (or a park, etc.) is intended to be placed at
2. Procedural generation of geometry on each of those buildings or zones, using a rule-based system based on CGA
3. During the previous generation, emit additional semantic information

The following subsections explain each of these steps in more detail.

3.3.1 Generation of city lots

Two main strategies to generate lots have been considered: a procedural generation which creates all the data, and importing GIS (Geographic Information System) real-world data.

3.3.1.1 Procedural lot generation

For this prototype, we were interested in having a basic city model for the purpose of focusing our efforts in the crowd generation part. Therefore road networks have not been fully implemented and instead we have developed a basic procedural lot generation.

The implemented simple generation algorithm creates rectangular isles of square lots, of a regular size, and whose dimensions can be tuned with some parameters (number of isles, number of lots per isle, etc.). For each isle, a random building type is assigned according to some probabilities read from a file. This generates a plain layout, but allows for variety on the resulting buildings distribution and hence for an heterogeneous crowd simulation.

3.3.1.2 GIS data loader

Another interesting feature for some use cases may be to use real-world geographic data as a basis to generate a virtual city, which in a future may be interesting in order to validate the population behavior. For such case, lots and road networks may be loaded from existing data.

For this purpose a loader for the OSM XML file format from OpenStreetMap [Ope16] has been implemented. This format consists on an XML file that includes a list of elements of the following types:

- *Node*: a 2D point as a pair of latitude/longitude coordinates
- *Way*: contains a list of references to nodes, forming polylines which may be open (sequence of segments) or closed (polygons). It is used to specify areas, roads, lots, buildings, etc.

- *Relation*: a kind of wildcard element that can reference nodes, ways, or other relations, and specifies semantics for this group. Examples of relations include defining bus routes, turning restrictions, elements forming a bridge, etc.

All of the previous elements contain *tags*, which are pairs of key/value entries. These specify what each element represents, or additional semantic data. For our purposes, the most interesting case is the *ways* that contain tags such as *building*, and *amenity*. These tags are used to specify regions that represent buildings, and indicate the kind of service they do offer.

OSM data can be easily downloaded from the OpenStreetMap [Ope16] website by just selecting a region. The main problem found with this generation mode is that the loaded data tends to be undertagged, i.e. most buildings are missing descriptors about their usage, which complicates the direct usage of such data. Moreover, stores or other locations are usually indicated as independent points that sometimes even appear outside the polygon data.

Some other notable issues are found regarding the quality of the polygons, which tend to include all kinds of problems such as self-intersections, reverse vertex winding order, "spikes" with almost-zero areas, etc. Even though some of these problems have been solved by some processing (remove repeated vertices, ensure the polygon normal points up, etc), some problems still remain and end up generating artifacts on the results.

Therefore, it is left as future work to study how to improve this data, both geometric and semantic, and how it can be supplemented by procedurally assigning additional semantics.

Figure 3.5 shows an example of loading the data shown in figure 3.4, which consists on a region of Barcelona downloaded from the OpenStreetMap website. The major problem that in the end prevented using this data for the simulation of a crowd is the difficulty on determining where to locate the entrances of the buildings (i.e. to determine the front facade) in a coherent manner, in part due to the aforementioned polygon problems.

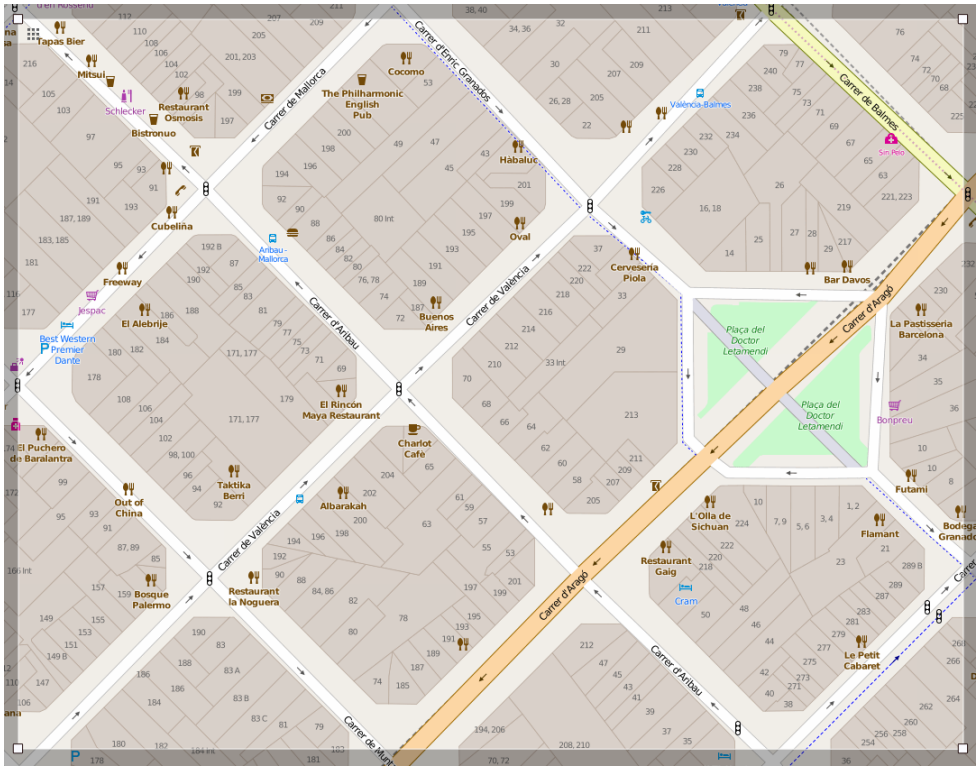

Figure 3.4: Example of exported data from OpenStreetMap website.

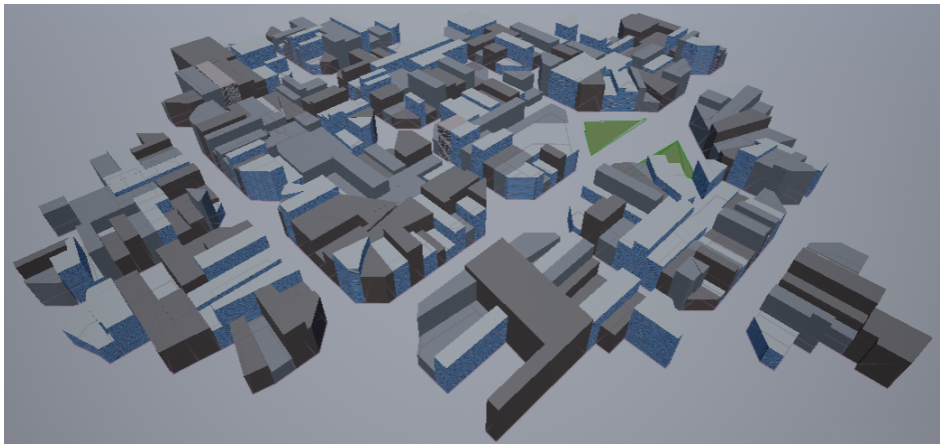

Figure 3.5: A very simple procedural generation on the data from figure 3.4.

3.3.2 Generation of buildings in the lots

To generate the geometry of the buildings, a custom implementation of a rule grammar interpreter has been developed, following the syntax and concepts from

CGA to allow compatibility with that files. More specifically, the syntax has been developed to be compatible with that of the CityEngine dialect, despite that our interpreter does just implement a subset of its operations, and it also does add some custom ones.

3.3.2.1 Rule file example

The core concept of this rule-based system are Rule Files. These files are usually identified by the .cga extension, and codify the set of rules that form a grammar. Figure 3.1 shows an example of such rule files.

```
// Attributes can be changed externally
attr buildingType = "house"

// A constant value is evalued only once
const floorHeight = 4
const numFloors = randInt(1, 3)


totalHeight = numFloors * floorHeight

// defines the start rule
@StartRule
Lot --> extrude(totalHeight)
        comp(f){ back: FrontFacade | top: Roof | all: Wall }

Roof --> taper(5) color(0.1,0,0)

FrontFacade --> split(x) { ~1: Wall | 2: FrontFacadeCenter | ~1: Wall }

FrontFacadeCenter --> split(y) { 3: Door | ~1: Wall }

Door --> extrude(-1)
         comp(f){ back: NIL | top: NIL | front: DoorInterior. | all : Wall }
         [ t('0.5, 0, '2) building(buildingType) ] // extension

Wall -->
  setupProjection(0, scope.xy, 1, 1)
  projectUV(0)
  texture("StarterContent/Textures/T_Brick_Clay_New_D")
```

Listing 3.1: CGA rule file example

First of all, some attributes can be specified with a default value which can be further overridden by assigning values to the *initial shape*. Also, constant functions may be specified (which are evaluated once for every generation), as well as non constant ones (evaluated each time).

The rules are specified as: `Symbol --> list of actions`. The actions may be either operations (e.g. `comp`, `extrude`, etc.) or symbols referring to other rule.

The state of an execution consists on a set of *shapes*, each of which has assigned a *symbol* (i.e. rule name) along with other attributes such as geometry, materials, etc. At each iteration, a *non-terminal shape* is selected and the rule corresponding to its symbol is executed. The actions on the right side of the rule are processed **sequentially and in order**, executing the operations, and creating new active shapes to be processed afterwards when symbols are found, by copying the current state at that time. If such symbol ends with a dot, a terminal symbol is created instead (i.e. it will not be expanded). Also, if a rule contains only operations, an implicit terminal shape is created at the end of it.

In the figure 3.1, a **Lot** rule is indicated as the starting point using an annotation `@StartRule`, and starts by transforming the 2D polygon to a 3D shape with the given height performing an `extrude` operation. Then, creates a different shape for some of its faces according to some selectors using a `comp` operation. This operation tries to match each component (e.g. f, which stands for *face*) in the current geometry with the provided selectors, and the contents of the first match are executed (if matched any).

For the **Roof**, it executes a `taper` operation, which creates a pyramid with the roof polygon as base and the given height. Then, sets it to a reddish color. As no symbol is specified, a terminal shape is then implicitly created.

The **FrontFacade** rule starts by dividing the facade horizontally. It sets a section of 2 meters in the center, and then divides the sides and sets them to form a wall. The tilde ($\sim$) indicates a *floating* value, i.e. a value that is computed to fill the remaining space after the fixed values are taken into account. The number is used as a weigh, in this case the same for both sides. Similary, **FrontFacade-Center** splits vertically creating a door of 3 meters filling the remainder with wall.

The **Door** rule extrudes with a negative height, creating a cube with inverted faces spanning inside the facade. It then removes the front side, creating an alcove. And the last line creates a building entrance (which is explained in the next subsection). But note the square brackets, which function as a push/pop pair of the current state and limit the effects of the operations that appear inside the brackets; as well as a translation operation `t`.

Finally, the **Wall** rule simply textures a polygon with a bricks texture. It creates a projection matrix to compute the UV coordinates *setupProjection*, creates the actual uv coordinates for the geometry with *projectUV*, and then assigns the texture name with *texture*.

The result of executing this rule can be seen in figure 3.6.

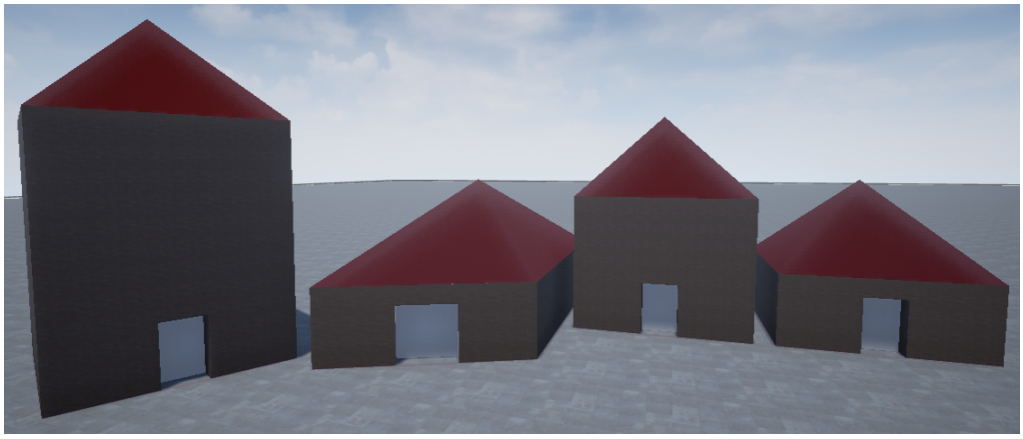

Figure 3.6: Example of results generated from the basic CGA rule file example

Two of the most important attributes of the shapes are the *scope* and *pivot*, which are transforms that define two coordinate systems (i.e. an origin point and an orientation for the three axes). The scope, which additionally has a size vector, can be thought of as a 3D box where a model (or a geometric primitive) will be fitted to. Most of the operations work on the scope coordinate system, but this scope-space is defined in pivot-space coordinates, which in turn are defined in object (or world) space. The division in different coordinate systems allows easily handling some operations: for instance, a component split may set the pivot coordinate system to a bottom corner of a facade of a building, and performing splits in this facade will create new scopes but keep a common pivot origin point for all new shapes.

For a more detailed explanation of the operations and their parameters, or the rule file syntax, the online manual of CityEngine [Esr16] may be consulted. Despite that not all operations listed there are supported by our current implementation, a large subset of them are. Moreover we have followed the same syntax and semantics so that the system can be easily extended in the future. Our current supported

subset contains the most common operations and functions, including all the ones listed in the previous example, with the exceptions of certain operations such as the ones to create specific shapes (L-shapes, U-shapes), or those used to generate more complex roof types than "taper".

3.3.2.2 Implementation details

We implemented a small parser for the rule files using the parser generator programs GNU Bison and Flex. This parser reads the rules, functions, etc. from the source file(s) into structures in memory, which are encapsulated inside a class `RuleFile`. Yet, this `RuleFile` class and its structures are not CGA-specific, and can be reused for other purposes, because the application of the rules depends on an external interpreter class. Those interpreter classes define among others the supported built-in operations, functions and values, as well as the attributes used as rule context. This decoupling of the rule file structure and the interpreter class allows us later on to reuse the same rule file format for the generation of the person agendas and tasks.

The actual generation is contained in a class called `RuleInterpreterCGA` which implements the CGA specific actions, functions, and shape, and is called using a `RuleFile` with a set of initial CGA shapes as input.

One of the main difficulties in the implementation of this part of the project was the geometric algorithms required. Contrary to the typical case in graphics processing, where we need to deal with triangle meshes, in our case it was necessary to deal with arbitrary polygonal meshes, whose polygons are not even guaranteed to be convex.

Some of the geometric processing operations implemented include:

- Split a given geometry in two by a plane, creating new vertices when necessary, and deal with the case that a polygon can be cut in more than 2 polygons if it is non-convex.
- Fill gaps on the geometry after this previous operation, which is performed by adding a face on the boundaries created by the previous step.
- Triangulation of 3D non-convex polygons, which is based on an *ear clipping*

algorithm [Ebe98]

- Extrusion of 3D polygons into volumes along a global or a per-face direction
- Shrinking/enlarging polygons by a given offset with respect to each edge, creating also new faces filling the (positive or negative) remaining space.
- Several other minor computations such as 3D polygon normals and areas, intersection of edges polygon, volume for 3D meshes (in part to check if a mesh is closed), and classification of components (faces/edges/vertices) according to different criteria such as orientation of their normals or other factors.

In addition, the implemented rule file parser and CGA interpreter can be executed as a standalone tool separated from Unreal Engine, which facilitates its integration with other systems.

3.3.3 Augmentation with semantics

During the generation of the buildings (and zones), two possible ways of adding semantics have been studied and incorporated in the system, expanding the geometric-oriented CGA capabilities.

The first way is that rules can be annotated with an `@Object("type")` tag, which results in the creation of a separated object with the specified tag. This is used for example to generate the benches on the parks and tag them as such. The goal of these new tags was to provide a prototype with easy integration of additional features. These new tags can be latter considered in the simulation to develop new tasks for the autonomous agents.

The second way consists of using URI (Universal Resource Identifiers) with a custom protocol part for the CGA instantiate `i("<uri>")` operations, which instead of loading geometry, they will cause Unreal Engine components to be instanced (still fitting the current scope when applicable). A class name is specified, along with a set of properties to be set, which can be assigned using UE4 built-in class reflection system that allows accessing the object properties dynamically by name.

A special operation `building("type")` has been also added, which internally re-

sorts to one of these special URIs. This operation defines in the world a building of the given type, creating an entrance point at the current CGA scope position. This offers greater flexibility as where to position the entrances to the buildings during the generation, and provides the basis to decide on topics such as the number of homes that fit in a building, or creating multiple "buildings" in each lot (e.g. a store on the first floor, and a house building entrance representing the upper floors).

Another operation `zone("type")` has been included, which defines a special area in the navigation mesh of the world. This allows for example to delimit zones of interest like parks, which the virtual agents may then use for various activities such as elder people going to sit on benches, or sporty people may visit to do jogging. Another application of this in a future may be to specify crosswalks in streets.

3.4 Generation of the population

One the city has been created along with its additional semantic information (building entrances, zones, objects, etc.), the next step is to generate a population that can behave accordingly based on the type of city and spaces that they will be interacting with. The goal of the thesis is on the one hand to ease the process of generating building with semantics to drive an autonomous population, and on the other hand to provide the user with enough flexibility to author and control the final emergent behavior. In this section we will explain how our autonomous citizens are created based on the city information obtained from the previous section.

3.4.1 Parse the city information

In a first step, the objects in the engine world are processed, performing a search for relevant components such as buildings, zones of interest, or interactable objects. This step is introduced instead of feeding directly the generated city information to the simulation manager, because it adds flexibility to the system: the user may want to manually edit the city after an automatic generation, or for example manually add special zones excluded from automatic generation to include unique landmarks.

Therefore, a small preprocess operation fills a structure with information about

the city and its buildings, which is needed latter on to generate the population and their agendas.

3.4.2 Generation of households and citizens

Instead of generating individual citizens, it resulted more realistic to create them in groups, which have been denominated as *households*. They basically consist of a set of persons which are understood to form a family, and whose relation is important to consider when assigning daily actions as for example parents taking little kids to school.

To generate the households, real world statistical data is used. In the tests and examples, a dataset from OpenDataBCN (Barcelona's City Hall Open Data Service) is used, which provides pairs of a semantic descriptor string of the household, and the number of occurrences. These counts are used to compute probabilities for the generation, and the semantic descriptors have been manually converted into household descriptors with a small converter script.

This processed data is stored as a JSON file, and a simplified example is shown in figure 3.2. The ID corresponds to the aforementioned semantic descriptor (in Catalan), followed by the count, and a descriptor for each member. The household members are then generated according to those descriptors, picking a household model each time using the corresponding probabilities.

```
{
  "households": [
    ...
    {
      "id": "UNADONADESETZEANYSIMESAMBUNOMESMENORS",
      "count": 13417,
      "members": [
        {
          "ageMin": 16,
          "female": true
        },
        {
          "ageMax": 17
        }
      ]
    },
    ...
    {
      "id": "DOSADULTSIUNMENOR",
      "count": 37611,
```

```
21        "members": [
22          {
23            "ageMin": 18
24          },
25          {
26            "ageMin": 18
27          },
28          {
29            "ageMax": 17
30          }
31        ]
32      },
33      ...
34    ]
35  }
```

Listing 3.2: Population JSON file example (simplified)

3.4.3 Generation of the citizens agendas

One the households and their members have been decided and created, we proceed to generate the individual agendas for citizens procedurally by executing a rule file. This generation creates as result a high-level schedule of the activities to be performed during the day, such as going to work or school and back home.

The agenda generation is started at household level: for each one, an execution of the rule file is started. Nonetheless, during this execution individual members can be "focused", which will cause all agenda modification operations to affect them and only them. This approach is more flexible than direct generation at person level, as it allows making decisions at household level such as who is to bring the kids to school (when applicable), or who should go shopping.

Although the agenda generation rule files follow the syntax of the CGA rule files and are stored in the same `RuleFile` class, they do not follow the semantics of CGA, and do in fact support a completely different set of operations and functions. A class `RuleInterpreterAgenda` handles the agenda generation by interpreting this other set of operations and rules. For example, in this interpreter there is no per-rule-instantiation state other than a set of variables and parameters, as opposed to CGA where it had attributes such as the scope, or the geometry. Instead, the state consists on a interpreter-global state that holds the current household and the focused member (if any), and any change to agendas or other person attributes will result in changes that are seen by any further rules or operations applied.

3.4.3.1 Built-in operations, functions, and variables

A unique characteristic of the operations and functions with respect to CGA rules is that in some cases one of the parameters may be a predicate, which is an expression that is not evaluated when the function is called, but evaluated inside the function using a different context in most cases. The main application of this is to evaluate expressions such as `age > 18` on a person-level to make decisions during the generation.

Additionally, most functions and operations make use of integer IDs that are interpreted as references to either persons, households or buildings.

Some built-in read-only variables are exposed that may be accessed to query attributes of the household or of the currently focused person:

- `home`: The ID of the building where the person (and household) lives
- `female`: A boolean which is true/false depending on the person gender
- `age`: A number representing the person age
- `person.id`: The ID of the currently focused person (or -1 if none)
- `household.id`: The ID of the current household

There are some functions that allow querying the environment and the city either for information, or to find element:

- `getDistance(<building1>,<building2>)`: Computes the nav-mesh distance between the entrances of the two given buildings
- `getDistanceInTime(<building1>,<building2>)`: Computes the expected time required for traveling between the two specified building entrances. Requires having a person focused, as it uses its walk speed to perform the computation.
- `findBuilding(<buildingType>)`: Returns a reference to a random building of the given type.
- `findNearestBuilding(<buildingType>,<buildingID>)`: Returns a reference to the building of the given type closest to the building provided as last

parameter.

- `findObject(<type>,<radius>?)`: Returns a reference to a random object of the given type. If a radius is provided, only objects within that distance from the current person are considered.
- `isValid(<itemID>)`: Returns a boolean indicating whether a building, object or zone ID is valid or not.

Some of the operations and functions are designed to query or produce effects at household level:

- `members { <cond1>: <actions1> | <cond2>: <actions2> | ... }`: this is the main operation at household-level. It implements the "focusing" feature previously mentioned. It works analogously to the CGA `comp` component split: i.e. for each member of the household, each of the conditions is checked sequentially, and the actions in the first match found (if any) are executed, having the current member focused. Conditions may be for example the age.
- `count(<predicate>?)`: counts the number of household members that match the specified criteria. If no predicate is given, returns the total number of members.
- `chooseMember(<predicate>)`: function that chooses and returns the ID of a member of the household among the ones matching the specified criteria. This function can be used for example to choose the adult from the household that can carry out a required action. The predicate may be also an heuristic scoring function, in which case the member that is evaluated with the largest value will be selected (or if multiple, one of them randomly).

The most important set of operations is aimed at inserting different new actions into the schedule, and most of them take as arguments the start (*t0*) and end (*t1*) times. For these operations to have an effect, a member of the household must be focused. Then the focused member will be the only one affected. These operations are:

- `stayInside(<t0>,<t1>,<buildingID>)`: instructs the person to stay "hidden" inside the given building
- `goToBuilding(<t0>,<t1>,<buildingID>)`: makes the person walk its way to the specified building, and enter it when reached.
- `accompany(<time>,<condition>)`: creates a shared group task where the

current person is the "leader", and other household members matching the condition join it. The task goal and start/end time are copied from the first non-leader member agenda, from the time specified as parameter. This operation is used in our examples to create tasks for a parent to accompany their children to school.

- `delayedRule(<t0>,<t1>,ruleName)`: This operation is one of the most interesting ones: it creates a delayed-evaluation order for the person to execute a rule as a task in the specified time slot. The specified rule will be used as starting point to dynamically generate behavior for the person. This allows taking into account factors such as resource availability (e.g. occupied park benches) and creates the basis for other emergent behaviors. The delayed rule executions have some particularities, and a slightly different set of possible rule operations; more on this is detailed in the next section.

Finally, there are some operations that are only supported during the execution of delayed-evaluation rules. They make use of a unique feature based on that: they can pause the execution until a given condition is meet, for instance after a certain time has passed. These operations are:

- `wait(<seconds>)`: keeps the person idle standing still for the specified duration
- `goToArea(<areaType>)`: finds and entrance to the nearest area of the given type, and makes the person walk to it, blocking the execution until it is reached.
- `goToObject(<objectId>)`: makes the person walk to the given object, and stalls execution until it is reached
- `sit(<objectId>)`: sits into the specified object (if that is possible).

3.4.3.2 Example of agenda generation rules

Figure 3.3 shows an example of a simple rule file featuring most of the previously described functions and operations, to implement some basic behaviors.

```
1
2 schoolStart = 8h
3 schoolEnd = 16h
4
5 workStart = 8h
6 workEnd = 16h
7
```

```
@StartRule
Household -->
  members { true: stayInside(0h, 24h, home) } // By default, everyone stay home
  members { age < 18:  ChildrenWeekday }
  members { age >= 18: AdultWeekday }
  [ // Have children? Then choose an adult to bring them to school
    case count(age < 18) != 0:
      members { chooseMember(age >= 18): BringChildrenToSchool }
  ]


VisitBuilding(t0, t1, building) -->
  set(time, getDistanceInTime(home, building))
  goToBuilding(t0 - time, t0, building)
  stayInside(t0, t1, building)
  goToBuilding(t1, t1 + time, home)

ChildrenWeekday -->
  set(school, findNearestBuilding("school", home))
  VisitBuilding(schoolStart, schoolEnd, school)

AdultWeekday -->
  set(workplace, findBuilding("workplace"))
  VisitBuilding(workStart, workEnd, workplace)

  delayedRule(18h, 20h, GoToPark)

BringChildrenToSchool -->
  accompany(schoolStart - 2, age < 18)

GoToPark -->
  wait(2)
  goToArea("park")
  set(bench, findObject("bench", 10))
```

Listing 3.3: Example of an agenda generation rule file

In this example, the execution starts at the **Household** rule, with no person focused. First of all, it applies some rules to all members (condition `true` means all pass), focusing them, one at a time, and setting a task to remain at home all day. Notice that this means adding new agenda entries may override previous ones, otherwise this line would make no sense. In case of a collision between agenda entries, the previous entries are split, trimmed, or deleted as needed to fit the new task.

The next two `member` actions in the **Household** rule select respectively the minors and the adults. In each case, they call a different rule. Both the called rules **ChildrenWeekday** and **AdultWeekday** are very similar, finding either a school or a workplace building (in case of school, the nearest to their homes), and then calling a helper rule **VisitBuilding** which computes the time required to travel to those places from home, and adds tasks to the agenda to go to the building, remain inside, and then come back home.

In the case of **AdultWeekday**, and additional action is added at then end, which adds an entry to their agenda for executing the rule **GoToPark** in a delayed manner. When the time comes, this rule will be executed, which will cause the person to stand still for 2 seconds, then walk to the nearest park entrance, and find a bench to sit in a 10m radius, and then sit on it. If no parks are found, or all benches are occupied, in this example the person would remain standing still doing nothing for 2 hours. To avoid this, the function `isValid` should be called to check the result of `findObject`, and an alternative course of action should be provided for the case it failed.

Going back to the end of the **Household** rule, there is an additional action: when the household has minors, it selects a random adult using `chooseMember`, and makes it accompany the children to school calling the rule **BringChildrenToSchool**. This rule calls `accompany`, which sets the adult leading a group containing all minors. It will copy the task of the first minor just right before the *schoolStart*, which will be a "go to school" task, therefore making the group go to the school. When that is accomplished, children will remain inside the building, but the adult will find that its next task is go to work, so they will head there. Notice that in this example, they only bring the kids to school, but do not pick them up; another `accompany` rule should be added for the school finish time.

3.5 Simulation of the citizens behavior

Once the city information has been gathered and the citizens and high-level agendas generated, the simulation can be started. This simulation keeps track of a time of the day, and updates the agents' agenda accordingly.

3.5.1 Initialization and time jumps

An important operation for the simulation consists of skipping the simulation to a specific time. This is an additional feature that has been implemented since the user may not interested in seeing the entire simulation of a full day, but instead just wants to see a specific time.

The simulation not only deals with running individuals' agendas and the whereabouts of the population, but it also includes trajectories and animations. Therefore the main issue when performing a jump in time is determining the exact location of each individual after the jump. This is currently solved by computing a normalized time value inside the current action (if moving from one location to another), so that 0 = beginning and 1 = end, and then using this value to compute an estimation of their expected position along a path in the navigation mesh between their origin and target locations.

Although this is not very accurate, as it assumes an optimal path at constant velocity and no other persons found in the path (i.e. no avoidance needed), it creates diverse initial conditions after one of such time jumps, which as the simulation advances tend to be corrected. In addition, to account for any of those unpredictable events, the expected traveling times computed are corrected to be slightly overestimated by increasing them by a fixed percentage.

3.5.2 Delayed execution of rules

When an agenda item was inserted with the operation `delayedRule(<t0>, <t1>, ruleName)`, the actual behavior decisions are delayed to the simulation runtime. When one person starts an agenda task of this type, a new instance of the *RuleInterpreterAgenda* class is created supporting most of the operations and functions described in the previous section, but notably without accepting agenda-modification operations (e.g. `stayInsideBuilding`, or other `delayedRule`). Instead, it supports a different set of low-level behavior-oriented operations, which corresponds to the last list in the *Built-in operations, functions and variables* section.

The major difference with respect to the executions done by the agenda-generation interpreter is that the execution in this case is **pausable** for some operations, and resumable later. This enables support for actions such as *waiting a specified amount of time*, or *until the person has reached a certain place* before continuing with the execution of the subsequent rules.

Therefore, rule actions are still executed sequentially, but some evaluations are delayed until the agent is prepared to act. All these features only affect the timespan of the `delayedRule` entry in the person agenda: i.e. when the ending time

is reached, it will interrupt any action and proceed to its next item in the agenda.

3.5.3 Implementation details

The simulation is integrated into the engine, as it requires access to building locations, citizen actors and their AI controllers among others, but part of the code has been separated into a base class to keep some degree of modularity. A base class `CitizenManagerBase` contains this logic, and most of the engine-dependent code is on a class `ACitizenManager` that is itself an actor (game object) for the engine.

3.6 Integration with the engine and editor

Part of the effort for this project has involved creating some editor extensions to allow us on the one hand to preview and debug the building generation, and on the other hand to visualize information of the simulation such as the current time, the generated persons, and their agendas.

3.6.1 Rule files and generation

A custom asset type has been created for the CGA rule files, which can be automatically imported into the editor, previewed, and edited; and which benefits from all the features of the assets in the editor (can be searched, easily assigned, etc.). In addition, a quick thumbnail preview has been implemented for these assets, as displayed in figure 3.7, which allows the user to do a quick scan to find the desired rule file.

The initial CGA shapes are represented by an object component class, which contains a reference to the rule file to use, initial rule, editable attributes, initial seed, etc., and an option to clear or regenerate the contents. This all is shown in figure 3.8. The attributes section is dynamically generated from the attributes read from the rule file, and some annotations on the attributes affect the control that is used to edit them: for instance, a `@Color` on a string attribute creates a color picker for

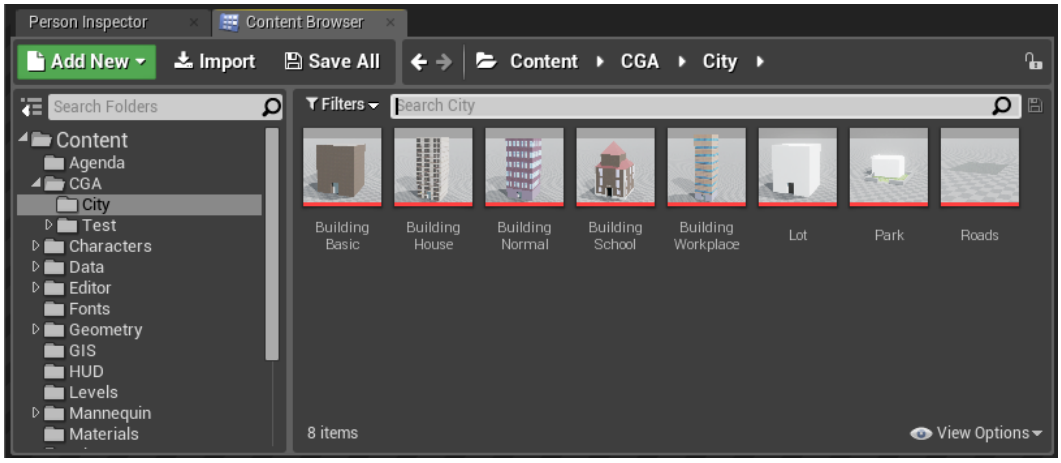

Figure 3.7: Content browser tab of the editor, showing the thumbnails implemented for CGA rule files assets.

it, or a `@Asset` annotation allows selecting a reference to a mesh or texture.

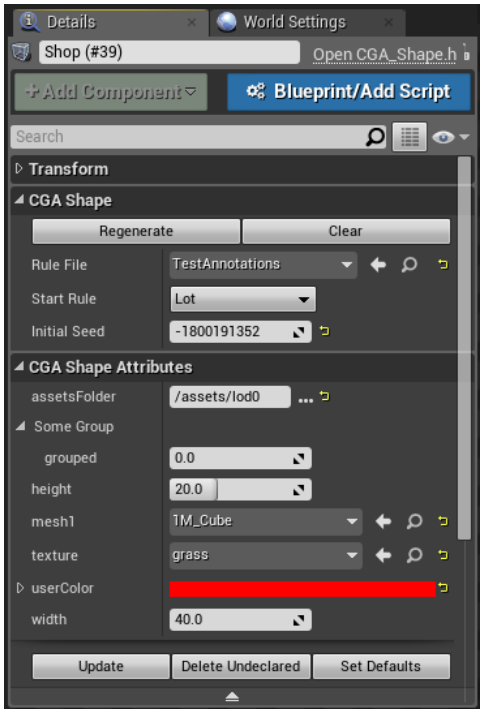

Figure 3.8: The properties panel for a initial CGA shape object.

In addition, an actor (game object) class containing this component by default has been created, but this class also inherits from a Brush engine class, and as such con-

tains a reference to geometry which can be edited using the built-in editor features.

Another component class has been created, which inherits from a class that allows handling a procedurally created mesh, and holds the results of the CGA generations. A custom visualizer for this component has been also implemented, which shows the pivot and scope position for each shape and hence can be used to debug the rule file. This can be used in combination with a debug option in the initial CGA shapes, which enforces generation of the full CGA shape tree as components, instead of the default behavior of grouping them together into just one or a few components. Both these options can be seen in figure 3.9.

In case of errors or warnings during the parsing or interpretation of the rule files, messages are printed into the editor console, and in case of errors, a message is also displayed on top of the screen for a few seconds.

Figure 3.9: Debugging features implemented for CGA rule files: on the left, the scope is shown for the shape (i.e. component) selected in the right.

3.6.2 Simulation

The Citizen Manager actor class handles the simulation of the time and the persons. It has some options that can be configured in the editor, such as the approximated number of persons to generate (the exact number depends on the households), the starting simulation time, or the population and agenda rule files to use.

To show the state of this class during the simulation, an editor window tab named *Person Inspector* has been implemented. In figure 3.10 we can see an example of the state to the *Person Inspector* for an instant of time of the simulation.

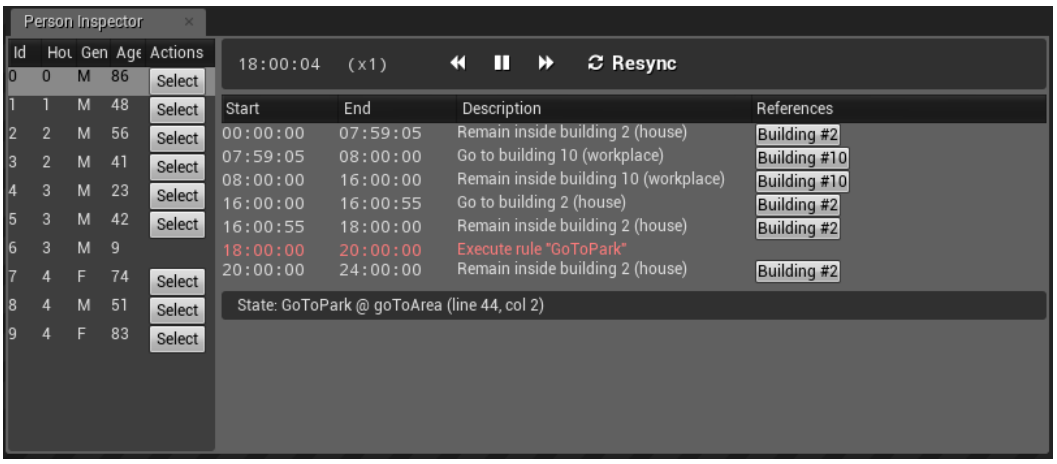

Figure 3.10: The implemented *Person Inspector* tab in the engine editor

On the left there is a list with all the persons, indicating their ID and the ID of their household, as well as their gender and age. It also provides a quick button to select the person actor, if it is spawned (i.e. if they are not inside a building). This along with the editor default shortcut of "go to actor" that centers the camera on the current selection, provides a quick way so see what they are doing.

On the right, at the top there is a control bar indicating the current simulation time, as the time of the day. And at the side, the current simulation speed, which may range between 0.25x and 20x (due to engine limitations). On its right side there are some control buttons to change this speed or pause the simulation.

The most important part is the right central section, which displays the agenda of the currently selected person. It lists all the entries on the day along with their start and end times, a description of the task performed, and a list of references to the buildings or other persons involved, which can be clicked to quickly select them. In addition, it highlights in red the task that is currently being performed.

Also on the right side, under the agenda, there is another bar, which is only visible depending on the type of the current task. The purpose of this bar is to show information on the current task state. In figure 3.10, it is visible because the cur-

rent task is the execution of a delayed rule. In such case, it displays information that includes the current call stack of rules, as well as the operation where the execution is paused at, and the line and column in the rules source file of this operation call.

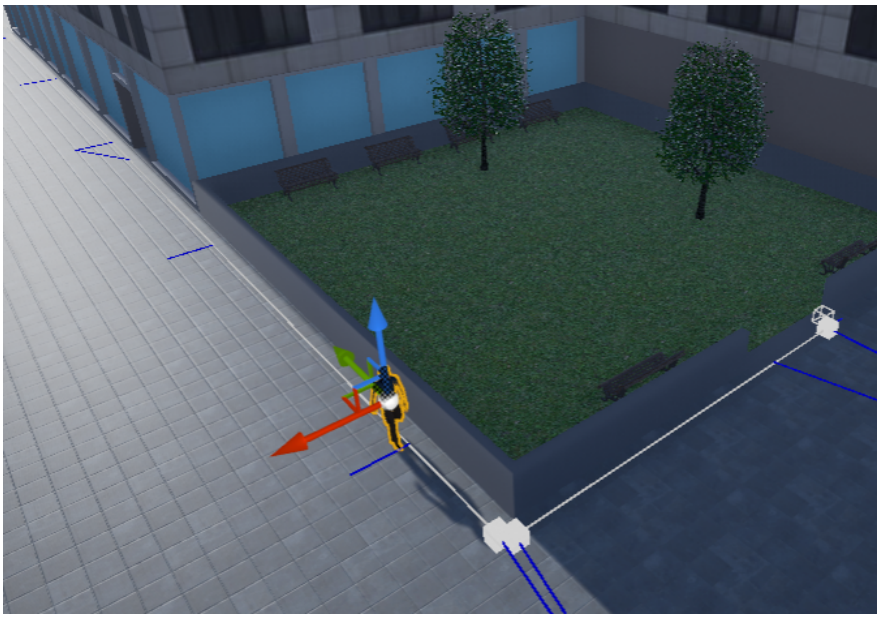

Figure 3.11: Debug visualization of the path a person is following. In this case, it is a result of a goToArea("park") operation.

During the simulation, also, if one or multiple persons are selected, the paths they are following are drawn in the world, as shown on figure 3.11, which allows keeping control of where each individual is headed to. In addition, the built-in unreal debug visualization for the RVO avoidance algorithm may be enabled from the console, which displays information about the application of that algorithm, such as when a agent has its velocity locked to avoid a collision.

Chapter 4

Results

4.1 Performance

We first want to note that performance is not a key element of this project, since our prototype focuses on providing a tool for authoring crowd behavior and simplifying the process of populating large virtual cities. So most of the work developed for this project is focused on pre-process activities. The online simulation depends strongly on aspects of the visualization, animation, and path-planing that are beyond the scope of this project. However in this section we will detail some performance information that may be helpful for the future development of the full system. The most important detail when it comes to performance, is the fact that we have succeeded at developing a system for generation and authoring of populated cities that allows for interactive response.

4.1.1 City and population generation

City and population generation are preprocess, and thus performance of this part of the project is not the main concern in this work. For this reason, no extensive testing has been done in the generation times for the city geometry nor the population. Note that this depends strongly on the different parameters indicated in this thesis, as well as on the engine being used. However, the generation time that we have observed during our experimental tests is on the order of a few seconds (under 7 seconds for the examples shown in next section).

The generation of the city geometry may last a few seconds depending on the number of lots to generate and complexity of the rules, as it involves also some work behind the scenes in the engine such as mesh and material optimization on their side, recompilation of shaders for materials, navigation mesh regeneration, scene tree reorganization, etc. Usually this time revolves around 5-6 seconds.

The generation of the city population in general is very fast, usually under one second, as it has no geometry involved and the rule interpretation is not very costly. Moreover, this process does not involve work on the engine whatsoever.

4.1.2 Simulation

As the simulation is a real-time execution, the performance to measure in this case are the frames per second. However, as could be expected, this value may be very volatile as it depends on the number of spawned persons in the world, which despawn when entering buildings. The number of people that is despawned also strongly depends on the generated schedules. Hence, in the virtual "rush hours" we can find the lowest FPS, that depending on the amount of people may reach non-interactive framerates.

In addition, the FPS depend also on many other factors such as the geometric complexity of the environment, or of the virtual agents model and animations, among others, and once again the engine internals, so the direct measure is very difficult and partially meaningless. The agenda execution algorithm can be timed, but times invested in this are very low, as most of the work lies on the low-level planning of the agents (path finding, collision avoidance, etc).

An obvious observation is that all of this will depend on the number of agents simulated. Some tests performed on a non-high-end machine (Core i5-4690k 3.50 GHz with 8 GB of RAM), where all the agents have been created with a same static agenda and hence all spawn at the same times, FPS greatly decreased at 1000 agents, whereas on the small hundreds it ran smoothly.

Another observation is that when having more groups of agents nearby, the FPS tends to decrease. This is generally due to the computation needed in the engine internals for collision handling between autonomous agents.

4.2 Examples

4.2.1 Generation of the city

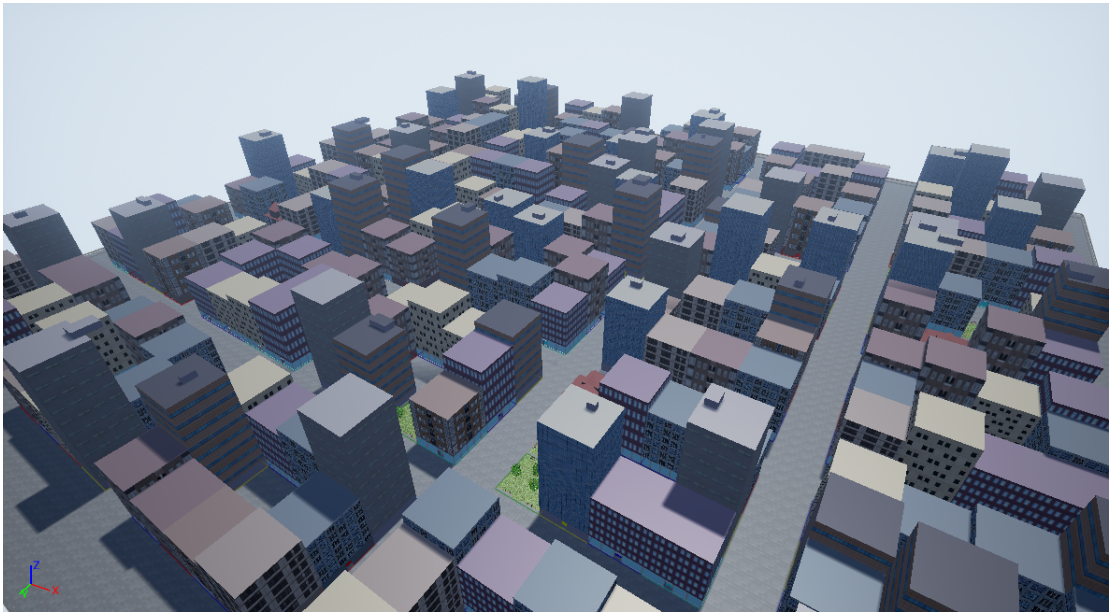

Figure 4.1: Example of a city created using the basic isle generation algorithm.

Figure 4.1 shows an example of a city generated using the system. Although there are only a few different rule files, combined with a few textures, the results in general present the appearance of high diversity, thanks to some detail variations such as the disposition of the trees in the parks, the varying number of floors, or the roof entrances for some buildings. There are some characteristics emphasized for the different building types: for example, office buildings tend to be taller, as opposed to residential buildings or shops. And these last two are very similar, but differ on the first floor (shops have display windows). Figure 4.2 shows a street-level picture of the city, notice how the shop entrances are not necessarily in the center, and the display windows adapt accordingly.

The main issue with the current resulting cities is the regularity of the lots sizes and distribution, all following the same pattern due to the lack of an advanced

city layout generation. This has been left as future work, due to time constraints, as the scope of this project was large and we decided to focus on the semantics generation and the population of the environment.

Despite the fact that the loading of GIS data to import lots is implemented and functional, these cities are not ready to be populated, not only because they still lack semantic information, but also due to the uncertainty on where to set up the building entrances. The imported data is very irregular, and as we said previously we can finds all kind of problems with it, for instance overlapping buildings, or a same building that is represented by multiple polygons. Therefore it is left as future work to study how to process such data globally in order to deduce valid and feasible locations for the entrances.

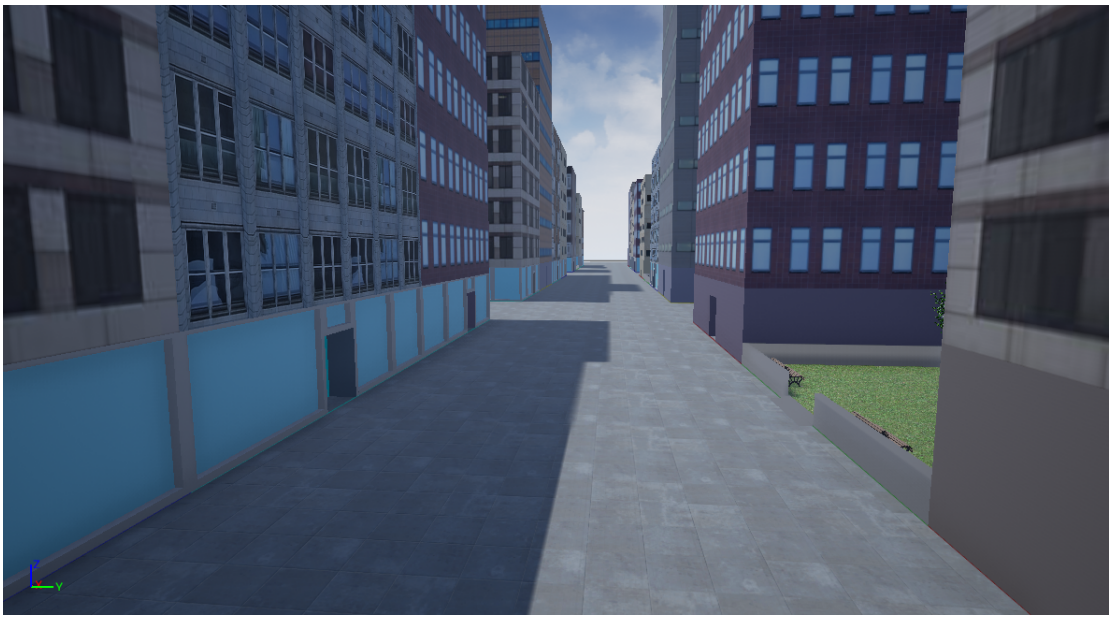

Figure 4.2: Street-level view of a generated city.

4.2.2 Simulation of the crowd

An example of a simulation is shown in figure 4.3, where in the morning people are going to their workplaces (e.g. the crystal building, third from the right). The doors of the buildings are colored according to the building type, for debugging purposes, as in some of the cases it may not be obvious.

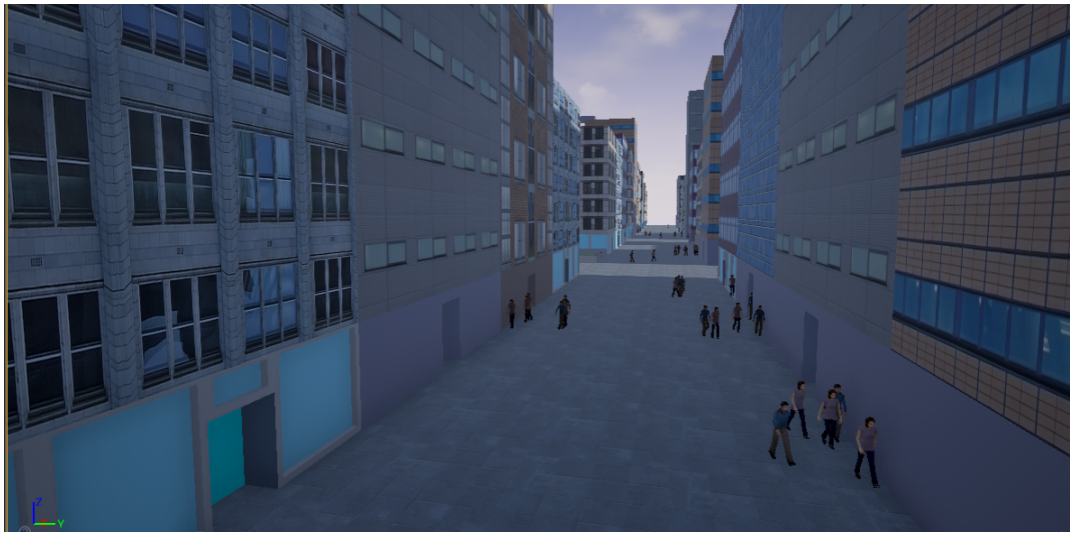

Figure 4.3: Simulation during the morning, where adults are going to their workplace.

In figure 4.4 we can see an example of the simulation when some parents are bringing their children to school. It also shows one of the main limitations of the current system: the default Unreal path-finding tends to find very direct paths, or paths that closely follow the building borders.

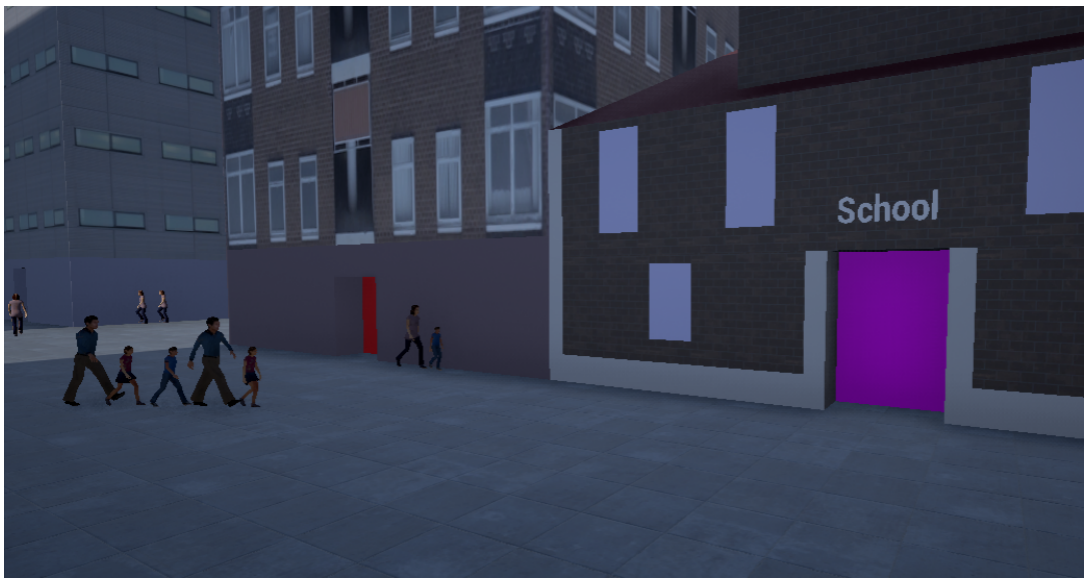

Figure 4.4: Simulation a few minutes before school start time, where some parents are accompanying their children to school.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, a prototype of a system for the procedural generation of cities and their populations has been designed and implemented. All of the required subsystems have been developed, but to different degrees, as the scope of this project is very large and it is intended to be continued in future work. We present our prototype as a proof of concept that procedural simulation techniques can be applied to authoring behaviors in large populated virtual cities. We have focused on the generation of the city buildings with embedded semantics, and on the generation of the virtual population and execution of the crowd simulation using the inhabitants' agendas.

In addition to the system that generates the populated cities, several debugging and visualization tools have been developed as extensions to the Unreal Engine editor. These tools will aid not only in future development of the project, but in the process of writing new rule files for both buildings and agenda generation, and in the experimentation and testing of the generation parameters.

All the goals that were planned for this master thesis have been successfully achieved, as the developed prototype is capable of generating the different components that were initially planned, as well as to run the simulation of the crowd of agents. The prototype has been integrated into the state of the art and popular Unreal Engine, and real-world data was studied and is in fact used during the generation of the households that form the population.

5.2 Future Work

The result of this project is a prototype which implements the core functionality for the system along with some basic features and behaviors, and provides the basis for future enhancement or extension of each of its individual modules. Therefore, there are a lot of directions in which this prototype has room for improvements.

5.2.1 City generation

The primary line of work in this part would be the generation of better city layouts (lots distributions). For this, the envisioned approach is the addition of road networks to generate a graph and partition space into blocks, to then divide blocks into lots. Some work on this direction was done, but it was dropped in favor of the central goals of the project due to time constraints.

Also, for a realistic city, generation of roads and streets should be added to system. For the generation of their geometry, these elements could be represented with the same component used for the lots. However, the addition of roads would entail also some issues, such as the need to block passage thought the pavement, and then adding crosswalks and street lights, and likely the need for a car simulation system.

On the generation and assignation of building types, a more realistic distribution should be studied, as the types of buildings nearby tend to be not independent in real life. However, real-world data modeling this is usually very difficult to find. During the study of a public real-world data, some data related to this topic was found. For instance, a list of lot isles of the city of Barcelona which includes in each case the number of locals, number of residences, and the total floor area. The inclusion of this or similar data in the process should be studied.

For the generation of the buildings, a subset of the CGA was implemented, which is missing some operations such as the generation of advanced roof geometry. These features could be also implemented to the system, or new useful operations could be analyzed for inclusion.

Another interesting topic would be the addition of a system that handles the evolution of a city, modifying the building and the semantic information and objects during the execution. In principle this would be easy to integrate with the current prototype, as it is already capable of regenerating buildings at runtime; the main issue would be ensuring it does not result in an impact on the framerate, as a building regeneration involves a partial regeneration of the navigation mesh or the re-cooking of the geometry in the Unreal Engine internals, among others.

5.2.2 Population generation and agent behavior

The data used for the generation of the population has a few issues, the main one being that for some families the number of male/female members is missing. Therefore additional data should be considered.

A major future line of work is to study the mutability of the agenda during the day, which may evolve as a result of the interaction of a person with other agents, or conditions of the environment (e.g. presence of roadworks blocking a street). Also, relationships between members of different households should be included, to model behaviors such as greeting friends on the street and stopping to talk to them.

A more complex personality system for the agents should be also implemented, as currently we only use a few attributes with randomly generated values. The influence that the agents relationships might have on their personality could be also studied.

For the low-level collision avoidance of the agents, we currently use Unreal's built-in RVO solution. However, we noticed several problems with this, including some issues in the resulting animation, and strange behaviors specially in the building entrances. Therefore, an alternative to this system or its modification should be considered, in order to achieve more realistic low-level trajectories. Similarly, the built-in path-finding should be adapted in order to avoid the agents following unnatural paths such as closely bordering the building facades.

5.2.3 Assets

The developed system is aimed to serve as a tool for authoring purposes, but nonetheless depends on some assets: mainly, the virtual agents models and animations, and the rule files used to generate the agendas and the buildings, as well as the assets used by those (facade textures, models such as for trees, etc.). Therefore, those assets would need to be improved with more variety than the ones used in the development of this prototype.

For the virtual crowd, more avatar models for different people ages should be added. In addition, the models textures should be augmented with mask maps that allow recoloring them at runtime, providing a efficient and easy way to increase the diversity in the crowd.

As for the generation of the agenda, more behaviors could be implemented in the rule files, but they may require also additional animations for representing their actions.

Chapter 6

Bibliography

[AB02] Jan Allbeck and Norman Badler. Toward representing agent behaviors modified by personality and emotion. In: *Embodied Conversational Agents at AAMAS* 2 (2002), pp. 15–19.

[ACC14] Pierre Allain, Nicolas Courty, and Thomas Corpetti. Optimal crowd editing. In: *Graphical Models* 76.1 (2014), pp. 1–16.

[All10] Jan M Allbeck. *Functional Crowds*. 2010.

[Aut16a] Autodesk. *3ds Max*. 2016. URL: `http://www.autodesk.com/products/3ds-max/overview` (visited on 04/30/2016).

[Aut16b] Autodesk. *Maya*. 2016. URL: `http://www.autodesk.com/products/maya/overview` (visited on 04/30/2016).

[Aut16c] Autodesk, Inc. *Autodesk Character Generator*. 2016. URL: `https://charactergenerator.autodesk.com` (visited on 04/30/2016).

[AWS08] Sharaf Al-kheder, Jun Wang, and Jie Shan. Fuzzy Inference Guided Cellular Automata Urban-growth Modelling Using Multi-temporal Satellite Images. In: *Int. J. Geogr. Inf. Sci.* 22.11-12 (Jan. 2008), pp. 1271–1293. ISSN: 1365-8816. DOI: `10.1080/13658810701617292`.

[Bas16] Basefount. *Miarmy - Better Crowd Simulation for Maya*. 2016. URL: `http://www.basefount.com/miarmy.html` (visited on 04/30/2016).

[BD16] Abdullah Bulbul and Rozenn Dahyot. Populated Virtual Cities using Social Media. In: *Computer Animation and Social Agents* (2016).

[Bea+11] Alejandro Beacco, Bernhard Spanlang, Carlos Andujar, and Nuria Pelechano. A Flexible Approach for Output-Sensitive Rendering of Animated Characters. In: *Computer Graphics Forum*. Vol. 30. 8. Wiley Online Library. 2011, pp. 2328–2340.

[BP13] Gonzalo Besuievsky and Gustavo Patow. Customizable LoD for procedural architecture. In: *Computer Graphics Forum*. Vol. 32. 8. Wiley Online Library. 2013, pp. 26–34.

[BP14] Alejandro Beacco and Nuria Pelechano. CAVAST: The Crows Animation, Visualization, and Simulation Testbed. In: *Spanish Computer Graphics Conference (CEIG)*. Ed. by Adolfo Munoz and Pere-Pau Vazquez. The Eurographics Association, 2014. ISBN: 978-3-905674-67-5. DOI: 10.2312/ceig.20141108.

[Cha+14] Panayiotis Charalambous, Ioannis Karamouzas, Stephen J Guy, and Yiorgos Chrysanthou. A Data-Driven Framework for Visual Crowd Analysis. In: *Computer Graphics Forum*. Vol. 33. 7. Wiley Online Library. 2014, pp. 41–50.

[Che04] Stephen Chenney. Flow tiles. In: *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association. 2004, pp. 233–242.

[Cry16] Crytek. *CryEngine*. 2016. URL: https://www.cryengine.com/ (visited on 06/30/2016).

[Dur+08] Funda Durupinar, Jan Allbeck, Nuria Pelechano, and Norman Badler. Creating crowd variation with the ocean personality model. In: *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*. International Foundation for Autonomous Agents and Multiagent Systems. 2008, pp. 1217–1220.

[Ebe98] David Eberly. Triangulation by ear clipping. In: *Geometric Tools, LLC* (1998).

[Emi+15] Arnaud Emilien, Ulysse Vimont, Marie-Paule Cani, Pierre Poulin, and Bedrich Benes. WorldBrush: Interactive Example-based Synthesis of Procedural Virtual Worlds. In: *ACM transactions on Graphics, Proceedings of ACM SIGGRAPH* 34.4 (2015), p. 11.

[Epi16] Epic Games Inc. *Unreal Engine 4*. 2016. URL: https://www.unrealengine.com (visited on 06/30/2016).

[Esr16] Esri. *Esri CityEngine*. 2016. URL: http://www.esri.com/software/cityengine (visited on 04/30/2016).

[Fen+16] Tian Feng, Lap-Fai Yu, Sai-Kit Yeung, KangKang Yin, and Kun Zhou. Crowd-driven Mid-scale Layout Design. In: 35.4 (2016).

[Gol+14] Abhinav Golas, Rahul Narain, Sean Curtis, and Ming C Lin. Hybrid long-range collision avoidance for crowd simulation. In: *Visualization and Computer Graphics, IEEE Transactions on* 20.7 (2014), pp. 1022–1034.

[Gol16] Golaem. *Golaem - Population tools for Maya*. 2016. URL: `http://golaem.com` (visited on 04/30/2016).

[Guy+10] Stephen J Guy, Jatin Chhugani, Sean Curtis, Pradeep Dubey, Ming Lin, and Dinesh Manocha. Pledestrians: a least-effort approach to crowd simulation. In: *Proceedings of the 2010 ACM SIGGRAPH/Eurographics symposium on computer animation*. Eurographics Association. 2010, pp. 119–128.

[Hen+13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural Content Generation for Games: A Survey. In: *ACM Trans. Multimedia Comput. Commun. Appl.* 9.1 (Feb. 2013), 1:1–1:22. ISSN: 1551-6857. DOI: `10.1145/2422956.2422957`.

[HFV00] Dirk Helbing, Illés Farkas, and Tamas Vicsek. Simulating dynamical features of escape panic. In: *Nature* 407.6803 (2000), pp. 487–490.

[Hoc+12] Rafael Hocevar, Fernando Marson, Vinícius Cassol, Henry Braun, Rafael Bidarra, and Soraia R Musse. From their environment to their behavior: a procedural approach to model groups of virtual agents. In: *Intelligent Virtual Agents*. Springer. 2012, pp. 370–376.

[Hon+04] Masanobu Honda, Kazunori Mizuno, Yukio Fukui, and Seiichi Nishihara. Generating autonomous time-varying virtual cities. In: *Cyberworlds, 2004 International Conference on*. IEEE. 2004, pp. 45–52.

[JL14] Carl-Johan Jorgensen and Fabrice Lamarche. *Space and Time Constrained Task Scheduling for Crowd Simulation*. Research Report PI 2013. Jan. 2014, p. 14.

[Jor+14] Kevin Jordao, Julien Pettré, Marc Christie, and M-P Cani. Crowd sculpting: A space-time sculpting method for populating virtual environments. In: *Computer Graphics Forum* 33.2 (2014), pp. 351–360.

[Jor+15] Kevin Jordao, Panayiotis Charalambous, Marc Christie, Julien Pettré, and Marie-Paule Cani. Crowd art: density and flow based crowd motion design. In: *Motion In Games*. 2015.

[Jor15] Carl-Johan Jorgensen. Scheduling activities under spatial and temporal constraints to populate virtual urban environments. Université de Rennes 1, 2015.

[JPC13] Kevin Jordao, Julien Pettré, and Marie-Paule Cani. Interactive techniques for populating large virtual cities. In: *Proceedings of the Eurographics Workshop on Urban Data Modelling and Visualisation*. Eurographics Association. 2013, pp. 41–42.

[JPC14] Kevin Jordao, Julien Pettré, and Marie-Paule Cani. Density-controlled crowds. In: *Symposition on Computer Animation*. 2014.

[Kap+13] Mubbasir Kapadia, Alejandro Beacco, Francisco Garcia, Vivek Reddy, Nuria Pelechano, and Norman I Badler. Multi-domain real-time planning in dynamic environments. In: *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM. 2013, pp. 115–124.

[Kap+15] Mubbasir Kapadia, Nuria Pelechano, Jan Allbeck, and Norm Badler. Virtual Crowds: Steps Toward Behavioral Realism. In: *Synthesis Lectures on Visual Computing: Computer Graphics, Animation, Computational Photography, and Imaging* 7.4 (2015), pp. 1–270.

[Kim+14] Jongmin Kim, Yeongho Seol, Taesoo Kwon, and Jehee Lee. Interactive manipulation of large-scale crowd animation. In: *ACM Transactions on Graphics (TOG)* 33.4 (2014), p. 83.

[KM06] George Kelly and Hugh McCabe. A survey of procedural techniques for city generation. In: *ITB Journal* 14 (2006), pp. 87–130.

[Mas16] Massive Software. *Massive*. 2016. URL: `http://www.massivesoftware.com/applications.html` (visited on 04/30/2016).

[Mix16a] Mixamo. *Adobe Fuse CC*. 2016. URL: `https://www.mixamo.com/fuse` (visited on 04/30/2016).

[Mix16b] Mixamo. *Decimator - Mixamo*. 2016. URL: `https://www.mixamo.com/decimator` (visited on 06/22/2016).

[Mix16c] Mixamo. *Mixamo*. 2016. URL: `https://www.mixamo.com/` (visited on 04/30/2016).

[MLF15] Francisco Martinez-Gil, Miguel Lozano, and Fernando Fernández. Strategies for simulating pedestrian navigation with multiple reinforcement learning agents. In: *Autonomous Agents and Multi-Agent Systems* 29.1 (2015), pp. 98–130.

[MT01] Soraia Raupp Musse and Daniel Thalmann. Hierarchical model for real time simulation of virtual human crowds. In: *Visualization and Computer Graphics, IEEE Transactions on* 7.2 (2001), pp. 152–164.

[Mül+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In: *Acm Transactions On Graphics (Tog)* 25.3 (2006), pp. 614–623.

[Nar+09] Rahul Narain, Abhinav Golas, Sean Curtis, and Ming C Lin. Aggregate dynamics for dense crowd simulation. In: *ACM Transactions on Graphics (TOG)*. Vol. 28. 5. ACM. 2009, p. 122.

[Ope16] OpenStreetMap. *OpenStreetMap*. 2016. URL: `https://www.openstreetmap.org` (visited on 06/22/2016).

[PA14] Cameron D Pelkey and Jan M Allbeck. Populating semantic virtual environments. In: *Computer Animation and Virtual Worlds* 25.3-4 (2014), pp. 403–410.

[PAB07] Nuria Pelechano, Jan M Allbeck, and Norman I Badler. Controlling individual agents in high-density crowd simulation. In: *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association. 2007, pp. 99–108.

[Pat+11] Sachin Patil, Jur Van den Berg, Sean Curtis, Ming C Lin, and Dinesh Manocha. Directing crowd simulations using navigation fields. In: *Visualization and Computer Graphics, IEEE Transactions on* 17.2 (2011), pp. 244–254.

[PM01] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 2001, pp. 301–308.

[RC10] Luis Henrique Oliveira Rios and Luiz Chaimowicz. A survey and classification of A* based best-first heuristic search algorithms. In: *Advances in Artificial Intelligence–SBIA 2010*. Springer, 2010, pp. 253–262.

[Rey87] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In: *ACM SIGGRAPH computer graphics*. Vol. 21. 4. ACM. 1987, pp. 25–34.

[SB10] Ben Sunshine-Hill and Norman I Badler. Perceptually realistic behavior through alibi generation. In: *AIIDE*. 2010.

[SW15] Michael Schwarz and Peter Wonka. Practical grammar-based procedural modeling of architecture: SIGGRAPH Asia 2015 course notes. In: *SIGGRAPH Asia 2015 Courses*. ACM. 2015, p. 13.

[SZP00] William Schuler, Liwei Zhao, and Martha Palmer. Parameterized action representation for virtual human agents. In: *Embodied conversational agents* (2000), p. 256.

[Uni16] Unity Technologies. *Unity - Game Engine*. 2016. URL: `http://unity3d.com` (visited on 06/30/2016).

[Val16] Valve Corporation. *Source (Game Engine)*. 2016. URL: `https://developer.valvesoftware.com/wiki/SDK_Docs` (visited on 06/30/2016).

[Van+09] Carlos A Vanegas, Daniel G Aliaga, Bedřich Beneš, and Paul A Waddell. Interactive design of urban spaces using geometrical and behavioral modeling. In: *ACM Transactions on Graphics (TOG)* 28.5 (2009), p. 111.

[VLM08] Jur Van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In: *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*. IEEE. 2008, pp. 1928–1935.

[Wad02] Paul Waddell. UrbanSim: Modeling urban development for land use, transportation, and environmental planning. In: *Journal of the American Planning Association* 68.3 (2002), pp. 297–314.

[Web+09] Basil Weber, Pascal Müller, Peter Wonka, and Markus Gross. Interactive geometric simulation of 4d cities. In: *Computer Graphics Forum*. Vol. 28. 2. Wiley Online Library. 2009, pp. 481–492.